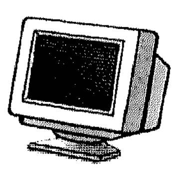
٢

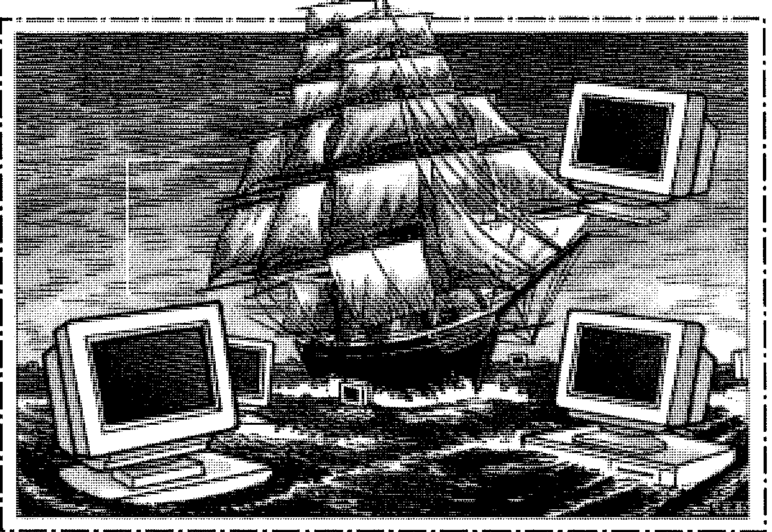
محاضرات كليبر

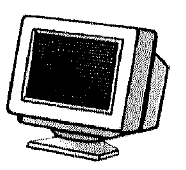
Clipper Course Notes

الجزء الثاني: أساسيات البرمجة

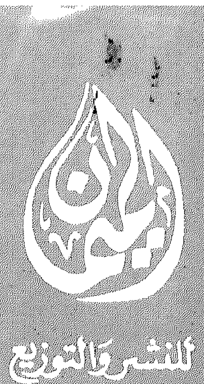
جديد !
الإصدار 5.2







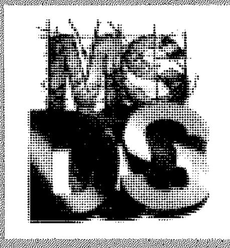




للنشر والتوزيع

في هذا الكتاب:

- شرح لمعظم الموضوعات الأساسية في كليبر 5.2
- تصميم وإنشاء وكتابة الرى التطبيقات باستخدام لغة كليبر
- يغطي المواضيع الخاصة باستخدام شبكة نوفيل مع كليبر 5.2.



سليمان اليمان

محاضرات كليبر

Clipper Course Notes

الجزء الثاني: أساسيات البرمجة

سليمان بن عبدالله الميمان

الدكتور/محمد سعيد دباس الأستاذ/أحمد فراس مهايني

النشر والتوزيع:

الميمان للنشر والتوزيع

ص.ب: ٩٠٠٢٠ – الرياض ١١٦١٣

هاتف: ٤٠٢١٢١٩ – ٤٢٦١١٩٤

فاكس: ٤٠١٤٩٩٦

محاضرات كليبر 5.2: أساسيات البرمجة

الطبعة الأولى – الرياض– ١٤١٥هـ

تنويه

تم إعداد هذه المحاضرات بالتعاون مع مؤسسة جرمبفيش الأمريكية المتخصصة في إعداد برامج تعليم لغة كليبر. وهذه المؤسسة معتمدة من قبل شركة Computer Associates، المالك الرسمي لجمع لغة كليبر 5.x .


بسم الله الرحمن الرحيم

المُحتَوَيَاتُ


تَمْهِيدٌ

تمهيد

الصفحات الموجودة بين يديك خلاصة تجارب وعمل امتدت عدة سنوات مع كليبر الإصدار 5.x فما فوق. وهو لغة برمجة متعددة الإمكانيات والقدرات تطورت بسرعة سنوات ضوئية متجاوزة الجذور المتواضعة للبرمجة باستخدام قاعدة البيانات dBASE ، وخاصة بعد إضافة كل من: المعالجات الأولية preprocessor ، والمتغيرات ذات النطاق المفرد ، والتعامل مع المصفوفات المختلفة array ، وكتل الشيفرة code blocks ، وفئات الهدف object classes.

وكما هي الحال ، لابد أن يلاقي كل أمر جديد بعض الصعوبات حتى يعتاد عليه الناس ويألفوه ، وليست البرمجة باستخدم كليبر استثناء من هذه القاعدة. وقد واجهت الكثير من الصعوبات أثناء التعامل مع لغة البرمجة هذه منذ ظهورها في أوائل عام ١٩٩٠ ، حيث لم نجد أحدا يعيننا ويقدم لنا المساعدة اللازمة. وهذا ما نريد، ونحاول أن نجنبك إياه ، وقد بذلنا كل جهد ممكن لإعداد هذا الكتاب الذي بين يديك ولم نألُ جهداً ولا وقتاً لتقديم العون، والمساعدة اللازمين لكل من يريد التعامل مع لغة البرمجة الرفيعة هذه.

إذا سبق لك –عزيزي القارىء– أن استخدمت كليبر Summer'87 ، ولم تستخدم كليبر 5 ، فستجد هذا الكتاب وسيلة مفيدة جدا لاغنى لك عنها. أما إذا كنت قد استخدمت كليبر ذاته إلى حد ما ، فإنك واجد قدرا كبيرا من المعلومات المفيدة في هذا الكتاب. ومهما كان مستوى خبرتك في هذا المجال ، فستجد نفسك بين الفينة والأخرى مضطرا للرجوع إلى فهرس الكتاب للتعرف على بعض المصطلحات والأوامر التي تلزمك وتحتاج لتنفيذها من آونة إلى أخرى لتطوير مختلف البرامج باستخدام كليبر 5.x.

وتعتبر المعلومات الواردة في هذا الكتاب متناسبة مع الإصدار 5.2 من كليبر ، كما سنشير في ثنايا الكتاب إلى أية معلومات جديدة مبنية على هذا الإصدار.

مفاتيح المجمع

يمكنك مجمع كليبر من استخدام من الخيارات والأوامر السطرية ، ومعظم هذه الأوامر جديدة في البرنامج. كما أن عددا منها هامة ، بل حرجة جدا ، خلال فترة التعليم ، كما ستجد أن هناك بعض هذه الأوامر ضرورية هي الأخرى خلال دراستك لتطوير برامج كليبر.

لاحظ أن كل مفاتيح المجمع هي حساسة للحالة (أي يجب أن تكتب كما هي تماما).

الخيار	الغرض
/a	إعلان متغير ذاكرة تلقائي.
/b	تصحيح المعلومات.
/credits	شاشة التوثيق.
/d<id>[=val>]	#define<id>
/i<path>	ممر البحث عن ملف #include .
/l	إخماد رقم سطر المعلومات.
/m	ترجمة ملف واحد فقط.
/n	لايتضمن بداية إجراء.
/o<path>	مشغل ملف الهدف/أو الممر.
/p	توليد ملف مخرجات للمعالج الاولي PRO .
/q	الخروج.
/r[<lib>]	يطلب من الرابط أن يقوم بالبحث عن <lib> .

الجدول مستمر من الصفحة السابقة....

الخيـــار	الغـــــرض
/s	التحقق من القواعد فقط.
/t<path>	مشغل/ممر الملفات المؤقتة.
/u[<file>]	استخدام البنية التعريفية للامر في <file> .
/v	المتغيرات المفترضة على النحو التالي: <-memvar.
/w	تمكين التحذيرات.
/x	ينتج قائمة من العلامات (Tokens) والاستدركات (Offset). (غير موثق undocumented).
/z	يعطل المختصرات المنطقية.

إعلان متغير الذاكرة الآلية /a

يرشد هذا الخيار الجمع إلى إعلان أي متغير تم تضمينه في أية عبارة من العبارات التالية: PARAMETERS, PUBLIC, PRIVATE, ليكون جزءاً من متغير الذاكرة MEMVAR.

ويمكن استخدام هذا الخيار لمنع التحير بين متغيرات الذاكرة ديناميكية النطاق وهي: (PUBLIC and PRIVATE) وحقول قاعدة البيانات. إلا أننا نفترض أنك ستأخذ اقتراحنا بصدد هذا الموضوع ، وهو أن تحذف إعلانات كل من: PUBLIC PRIVATE من برامجك ، ولن يكون هناك أي داع لاستخدام مفتاح a/.

وهناك أمر متعلق بهذا أيضا نود أن نقترحه على مستخدمي "كليبر" ، وذلك إذا لم يكن هذا الاقتراح قيد الاستخدام حاليا، وهو: يجب أن تشير إلى الحقول دائما ، بأن تسبقها بالاسماء المستعارة المطابقة لكل منها :

```
USE customer new
mname := Customer->lname        // جيد
mname := lname                  // سيء
```

ولايقتصر هذا الأمر على توضيح كل من الحقول ، وغير الحقول ، بل يوفر وقتك الثمين لصيانة برامجك بحيث يمكنك أن تتعرف خلال لحظات قليلة على منطقة العمل التي يطابقها ذاك الحقل المطلوب. أما إذا فتحت عدة مناطق عمل فسيبين لك هذا الفرق الواضح بين "اكتشاف الخطأ البرمجي بسرعة" ، وبين الصداع الحقيقي الذي يمكن أن ينشأ عن ذلك البحث.

اكتشاف الخطأ البرمجي وتصحيحه /b

يتضمن "كليبر" 5.x برنامجا لاكتشاف الأخطاء وتصحيحها (Debug) على مستوى المصدر ، يمكنك من اكتشاف الأخطاء البرمجية وتصحيحها بشكل سريع ، بحيث يمكنك من مشاهدة شيفرة المصدر أثناء تنفيذ البرنامج. إلا أن هذا سيتطلب إعداد برامجك بطريقة تختلف شيئا ما عن الطريقة التقليدية ، وذلك إذا أردت أن تقوم باستخدام برنامج اكتشاف الخطأ البرمجي وتصحيحه. فإذا أردت اكتشاف الأخطاء البرمجية الموجودة في تطبيقات CA-Clipper 5.x ، يجب أن تضمن مفتاح /b في هذه التطبيقات ، والذي يقوم بوضع "معلومات اكتشاف الخطأ البرمجي وتصحيحه" في ملف البرنامج المطلوب. كما يجب أن تترك أيضا أرقام الأسطر في ملف البرنامج المطلوب. أو ، بمعنى آخر: إذا أردت استخدام خيار /b فيجب ألا تستخدم خيار /l على الإطلاق.

إن استخدام خيار b/ سيضيف حوالي (٥-٧) بايت إلى ملف البرنامج المطلوب ، والمعروف بالنهاية:(.OBJ) لك سطر من أسطر شيفرات المصدر ، بحيث يصبح الملف التنفيذي لهذا البرنامج أكبر نسبيا من غيره. إلا أن حجم الملف التنفيذي هذا (.EXE) ليس ذا أهمية هنا ، وذلك الرابط RTLINK. يضع شيفرات كليبر تلقائيا داخل إحلال ديناميكي. لذا ، فيستحسن أن تأخذ بعين الاعتبار استخدام الخيار b/ بشكل مستمر. (ويحتمل أنه خلال عملية التعرف على عمليات "كليبر" 5.x ستقوم بإجراء كثير من عمليات اكتشاف الأخطاء البرمجية وتصحيحها).

إذا أردت أن تقوم بعملية اكتشاف الأخطاء البرمجية وتصحيحها بشكل اختياري على البرامج التي تعدها ، فربما يمكنك تجميع بعض الوحدات البرمجية فقط باستخدام الخيار b/ . ثم لدى تشغيل البرنامج مع استخدام برنامج اكتشاف الأخطاء البرمجية وتصحيحها Debug ، فإن هذا البرنامج سيتوقف فقط عند الوحدات التي تم تجميعها باستخدام الخيار b/.

خيار شاشة التوثيق والاستحقاق credits/

تعرض هذه الشاشة اسماء كافة الأشخاص الذين قاموا بتطوير هذه البرامج باستخدام كليبر 5.x وتنفع هذه الشاشة بشكل خاص إذا أردت أن تثني على جهود بعض الأشخاص وتعترف بها ، أو تلوم أحدا على فعله.

الخيار <d<id>[=<val>] #define/

يحدد هذا الخيار ثابت ظاهريا للمعالج الأولي. وتمثل العبارة <id> اسم ذاك الثابت، ويمكن أن تعين قيمة الثابت <val> اختياريا ، بأن تتبع <id> بإشارة = ثم تحدد القيمة المطلوبة.

إن الثوابت الظاهرية manifest constant هي إشارات (علامات) (flags) توضع خصيصا من أجل المعالج الأولي. ويتم تعريف هذه العلامات في ملف (.PRG) باستخدام

موجه التعريف (#deifne). وباستخدام كل من توجيهي: #ifdef و#ifndef سيتم تضمين المعالج الأولي ، أو يتجاهل الأقسام من شيفرة المصدر بناء على وجود الثابت الظاهري أو عدم وجوده.

إن الخيار d/ يسهل عملية التجميع المشروط إلى حد كبير ، وذلك لأنه يمكنك تحديد #define الثوابت الظاهرة على سطر الأوامر بدلا من تغيير شيفرة المصدر. وسنبين هذا الأمر بمزيد من التفصيل لدى الحديث عن المعالج الأولي preprocessor.

الخيار ES/ اخرج من الخطورة

بما أن تحذيرات المجمع غالبا ما تكون دلالة على كارثة محتملة أثناء وقت التشغيل ، فإن القدرة على تجهيز المجمع على مستوى "اخرج من الخطورة" هي أمر مفيد جدا لإنهاء جلسة عمل RMAKE بدلا من الاستمرار في دورة الربط. وتمكنك الخيارات التالي الجديدة من تحقيق هذا الأمر:

- **ES/** : هذا هو مستوى "اخرج من الخطورة" المفترض ، وهو يتطابق كليبر 5.x فإذا تمت مواجهة أية تحذيرات أثناء عملية التجميع ، فإن المجمع لايقوم بتجهيز خطأ "دوس" لدى الخروج من البرنامج.
- **ES0/** : هذا الخيار هو مساو للخيار **ES/**.
- **ES1/** : يحدد هذا الخيار مستوى "اخرج من الخطورة" على المستوى الأول. فإذا تمت مواجهة أية تحذيرات أثناء عملية التجميع ، فإن المجمع يجهز مستوى خطأ "دوس" لدى الخروج من البرنامج. (ولم يكن هذا الخيار موجودا في إصدار كليبر **5.0x**).
- **ES2/** : يحدد هذا الخيار مستوى "اخرج من الخطورة" على المستوى الثاني. فإذا تمت مواجهة أية تحذيرات أثناء عملية التجميع ، فإن المجمع يجيب على ذلك بعدم تجهيز ملف (**.OBJ**).

ملاحظة هامة

تتطلب عملية التجميع باستخدام كليبر 5.2 إلى امكانية معالجة ٢٥ ملف معاً كحد أدنى. لذلك ، تأكد من أن عبارة FILES في ملف إعداد نظام التشغيل DOS الخاص بك config.sys مضبوط على FILES=25 على الأقل. فإذا كنت تعمل على شبكة نوفيل NOVELL افعل الشيء ذاته لعبارة معالجة الملفات في ملف SHELL.CFG الخاص بك.

خيار تضمين مسار البحث عن ملف #include /i<path>

يلحق هذا الخيار الدليل المحدد بحيث يضعه أمام قائمة المسار المحددة باستخدام متغير البيئة INCLUDE. ولايقتضي هذا الخيار إضافة الشرطة المائلة العكسية (\) إلى اسم المسار. ويحدد سطر الأوامر التالي دليلا إضافيا ، C:\INCLUDE بحيث يتم البحث فيه عن ملفات الترويسة.

```
clipper myprog   /ic: \include
```

ويمكن تحديد عدة مسارات بحث إذا لزم الأمر عند الضرورة وذلك باستخدام قائمة تبدأ بفاصلة منقوطة ;. فعلى سبيل المثال: سيقوم الأمر التالي بالبحث في دليلين إضافيين هما: C:\APPS و C:\INCLUDE للملفات الموجودة في الترويسة.

```
clipper myprog  /ic :\apps; c :\include
```

ومع هذا ، فستجد في كثير من الحالات بأنك تكتفي بإبقاء ملفات ترويساتك جميعها (.CH) في دليل واحد. والاستثناء الوحيد لهذا الأمر هو أن يكون لديك ملفات ترويسات خاصة ومحددة لبعض البرامج.

وعندما تقوم بتجميع برنامج ما ، فإن المعالج الأولي في كليبر 5.x سيقوم بالبحث أولا عن ملفات الترويسة في الدليل الحالي ، ثم في أي دليل آخر محدد في الخيار i/ وأخير في أي دليل محدد بواسطة متغير البيئة INCLUDE.

فكرة مفيدة

سيقوم المعالج الأولي بالبحث في ملفات الترويسة التي تظهر أولا ، وبناء على ذلك ، فإذا كان لديك نسخة محدثة من ملف الترويسة ، واحدة في الدليل الحالي وأخرى في الدليل المعتاد INCLUDE\ ، فإن المعالج سيتجاهل الأحدث ويبقي النسخة الأقدم.

خيار إخماد أرقام السطور l/

يقوم هذا الخيار بإزالة أرقام سطور برنامج شيفرة المصدر من ملف الهدف. وهذا سيوفر ثلاث بايت لكل سطر من سطور شيفرة المصدر. وبناء على ذلك ، إذا قمت بتجميع ١٠٠٠ سطر من شيفرة المصدر باستخدام الخيار l/ ، فإنك ستوفر ٣٠٠٠ بايت في الملف التنفيذي EXE..

وكما ذكرنا أعلاه عند الحديث عن الخيار b/ ، فإننا ننصح بتحاشي استخدام الخيار l/ وذلك لأن الحجم والذاكرة المستخدمة أقل بكثير من التي يستخدمها الرابط RTLINK..

وهناك عامل آخر هام وهو عندما يتحطم البرنامج ويتوقف عن العمل كلياً، فإن نظام رسائل الأخطاء في كليبر سيرجع بهذه الطريقة رقم السطر (وهذا أفضل من أن لايرجع شيئا عندما تستخدم الخيار l/). وهناك احتمال كبير بأن البرنامج الذي تقوم بتطوير سيتحطم كثيرا أثناء التطوير بكليبر ، ولذلك ينبغي أن تبذل قصارى جهدك لتوفير كل الوسائل الممكنة للحصول على أكبر قدر من المعلومات للتقليل من مشاكل التصحيح.

خيار تجميع وحدة واحدة فقط m/

يقوم هذا الخيار بتجميع ملف البرنامج PRG. الحالي فقط ، ويخمد البحث التلقائي عن أية ملفات PRG. أخرى مشار إليها في ملف البرنامج بأي من الأوامر التالية DO أو SET FORMAT أو SET KEY أو SET PROCEDURE. فإذا قررت استخدام الوظائف بدلا من الإجراءات ، عندئذ يصبح استخدام المفتاح m/ غير ضروري.

وكما ذكرنا عندما تحدثنا عن ملفات CLP. ، فإن برنامج الربط RTLINK. يقوم تلقائيا بإزالة التكرار من جدول الرموز جاعلا من الخيار m/ أكثر إغراءً. وفي الحقيقة فليس هناك أي داع لتجميع عدد من ملفات PRG. في ملف هدف واحد OBJ..

خيار إخماد إعلان بداية إجراء n/

يقوم هذا الخيار بإخماد التعريف التلقائي لإجراء يحمل اسم ملف البرنامج PRG. ذاته. ينبغي دائما استخدام هذا الخيار ، وتجعل من العادة الدائمة لك أثناء البرمجة أن تستهل أول وظيفة في برنامجك PRG. باستخدام العبارة FUNCTION.

لماذا نقوم بهذا المجهود ؟ إن السبب هو وجود المتغيرات الساكنة في الملف الواسع file-wide ststic variables. لنفترض أننا نرغب في أن تكون المصفوفة _SETTINGS مرئية لكل الوظائف في ملف البرنامج SETUP.PRG ، يمكننا عمل ذلك بإعلان _SETTIGS كمتغير ساكن قبل أول وظيفة في ملف البرنامج.

```
// SETUP.PRG
static settings_ := { } // visible in both MAIN( ) and MODIFY( )

function main
local x
for x = 1 to 5
   aadd(settings_, x)
next
modify( )
aeval(settings_, { | a | qout(a) } )
```

```
return nil

function modify
local x
for x = 1 to len(settings_)
  settings_[x]++
next
return nil
```

إذا قمت بالتجميع دون استخدام الخيار n/ ، فإن المجمع سينشيء إجراء استهلالياً بعنوان SETUP. وسيحتوي هذا الإجراء على الإعلان الساكن static declaration قبل الوظيفة ()MAIN. وهذا العمل لن يقدم لك أي شيء عند التشغيل سوى إضاعة الوقت.

أثر جانبي آخر مفيد للخيار n/ هو أنك تحفظ الذاكرة في الملف التنفيذي EXE.. كل وظيفة وإجراء لابد أن يمثل اسمها بإدخالها في جدول العنونة بالإضافة إلى جدول الرموز. إذا قمت بالتجميع باستخدام الخيار n/ لإزالة وظيفة الاستهلال غير الضرورية، فإنك ستحفظ بذلك مالايقل عن ١٢٥ بايت لكل وحدة هدف object module. وإن كانت هذا القدر لايعتبر كبيرا ، ولكن كما يعلم غالب المطورين لبرامج كليبر ، بأنه عندما تكون المسألة تتعلق بالذاكرة ، فإن القليل منها قد يفيد ويساعد.

تذكر

إذا كنت تخطط لاستخدام المتغيرات الساكنة STATIC على مدى الملف ، فتأكد بأنك تقوم بالتجميع باستخدام المفتاح n/. وإذا لم تتذكر عمل ذلك ، فإن ذلك سيقوم برنامج أثناء التشغيل إلى مشاكل ليس لها نهاية مما يجعلك في حيرة من أمرك. تعود من الآن فصاعداً على استخدام المفتاح n/ أثناء التجميع ، وذلك لأنك ستستخدم المتغيرات الساكنه بكل تأكيد للاستفادة القصوى من إمكانيات كليبر الكاملة.

خيار مشغل ملف الهدف أو الممر /o<path>

يعرف هذا الخيار اسم و / أو الموقع المطلوب لملف الهدف المخرج. يوضح المثال التالي تجميع البرنامج MYPROG.PRG إلى BLAHBLAH.OBJ ويضع ملف الهدف الناتج في الدليل C:\OBJ.

```
clipper myprog /oc:\obj\blahblah
```

وفي حالة تحديد دليل غير موجود ، ينتج عن ذلك خطأ قاتل fatal error وبالتالي ستتوقف عملية التجميع للبرنامج myprog.

ملاحظة
إذا كنت ترغب في تحديد الممر path فقط ، فإنه ينبغي عليك إنهاؤه بالشرطة المائلة العكسية ("\").

خيار توليد ملف مخرجات المعالج الأولي /p

يقوم هذا الأمر بالإيعاز للمجمع بنسخ مخرجات المعالج الأولي لملف PPO.. وسيحمل هذا الملف اسم ملف PRG. ذاته، وليس هناك أي طريقة لإعطائه اسماً آخر للملف. ونحن ننصح باستخدام هذا الخيار للأغراض التعليمية.

الأمر التالي أدناه يقوم بإنشاء الملف MYPROG.PPO :

```
clipper myprog /p
```

افحص ملف PPO. لتشاهد كيف تبدو شفرتك حقيقة في المعالج الأولي وكذلك المجمع. إن هذه الطريقة توفر أسلوباً رائعاً وممتازاً لتعلم الكثير عن العمل الداخلي لكليبر 5.x.

خيار الخروج q/

يقوم هذا الخيار بإخماد عرض أرقام السطور أثناء عملية التجميع ، وبذلك يمكن أن يوفر لك عدداً من الثواني أثناء تجميع ملفات البرامج الطويلة.

خيار البحث عن المكتبات r[<lib>]/

يقوم هذا الخيار بتثبيت (أو إزالة) طلب البحث عن المكتبات داخل ملفات الهدف obj.. وعندما تقوم بالربط يتم البحث عن المكتبات لإعادة حل أية إشارات لم يتم حلها في وقت التجميع. فعلى سبيل المثال ، إذا حاولت تجميع البرنامج التالي وربطه:

```
function  main
myfunc( )
return nil
```

سيتم إجبار برنامج الربط أن يبحث في مكتبات كليبر (CLIPPER.LIB و EXTEND.LIB و TERMINAL.LIB و DBFNTX.LIB) للبحث عن رمز MYFUNC.

يقوم مجمع كليبر 5.x بالتثبيت التلقائي لطلب البحث عن CLIPPER.LIB و EXTEND.LIB و DBFNTX.LIB و TERMINAL.LIB داخل ملفات الهدف object files. وهذا يعني أنك لست بحاجة لذكر اسماء هذه المكتبات في أمر الربط (على افتراض أنك قمت بتحديد متغير البيئة LIB بحيث يمكن للرابط أن يجدها).

إن استخدام الخيار r/ سيلغي عمل هذه الافتراضات ، إلا أنه قد يسبب بدوره آثاراً جانبية كأن تظهر عشرات الرموز غير المعروفة أثناء "زمن الربط " ، و "الموت المفاجيء" أثناء عملية التشغيل.

إذا استخدمت الخيار r / دون استخدام المعلم (LIB) ، فلن يتم تضمين أي من طلبات البحث. ويمكنك أيضاً تحديد الخيار r/ عدد من المرات بحيث تضمَّن البحث عن أكثر من مكتبة واحدة ، كما يبين سطر الأوامر التالي:

Clipper myprog /rGRUMP /rMYLIB <-- Search GRUMP.LIB & MYLIB.LIB

خيار فحص التركيب اللغوي s/

يمكنك هذا الخيار من فحص التركيب اللغوي للملف البرنامج PRG. دون إعداد وإنتاج أي ملف هدف **object file.**

Clipper myprog /s >> error.txt <-- Write any errors to ERROR.TXT

خيار (السواقة/أو المسار للملفات المؤقتة)/t<path>

يتيح لك هذا الخيار تحديد دليل آخر للملفات المؤقتة التي يتم إنتاجها خلال التجميع وستسرع هذه العملية التجميع عندما يكون لديك حجم كاف من الذاكرة العشوائية للقرص. ويستخدم المثال التالي الذاكرة العشوائية المتوفرة في السواقة **D:** لتخزين الملفات المؤقتة:

Clipper myprog / td :

تحذير

إذا استخدمت خيار t/ للتجميع ، فإن السواقة والمسار اللذين يشار إليهما ليسا موجودين فقط ، إلا أن فيهما حجماً كافياً لاستيعاب أكبر ملف برامج PRG. تريد تجميعه ، وإلا فستكون ضحية لظهور الرسالة التالية : "cannot create intermediate file") لا أستطيع تجهيز ملف وسيط) ، وهي رسالة خطأ تجميع معروفة.

خيار (استخدم تعريفات الأمر البديلة) /u[<file>]

يوجه هذا الأمر المجمّع إلى استخدام ملف ترويسة قياسي بديل لإجراء عمليات المعالجة الأولية لشيفرة المصدر ، بدلاً من الافتراضات الموجودة في ملف STD.CH (والتي تم تضمينها ذاتياً ومباشرة في الملف التنفيذي CLIPPER.EXE). وسيبحث برنامج المعالج الأولي ، أولاً في الدليل الحالي ، ثم في أي دليل آخر يتم تحديده باستخدام متغير البيئة Include.

تحذير

إذا استخدمت هذا الخيار ، كن مستعداً لتحاسب عن كافة أوامر كليبر الموجودة في ملف الترويسة البديل القياسي ، إذ أنه سيتم تجاهل الملف المفترض STD.CH كليا.

إذا أردت إنشاء مجموعة أوامر خاصة بك ، فإننا نقترح أن تعدّ أولاً نسخة من ملف STD.CH ، ثم تغير اسمها ، وتحرر الملف، أو تعدله. ثم يمكنك بعد ذلك تحديد ملف CH. الجديد باستخدام الخيار u/. وسيستخدم المثال التالي تعريفات أوامر بديلة يحتويها ملف الترويسة ARABIC.CH.

```
Clipper  myprog  /uarabic.ch
```

الخيار /v (يفترض أن تكون المتغيرات ->MEMVAR)

يوجه هذا الخيار المجمّع إلى أن يفترض أن تكون كافة الإسنادات لمتغير غير معلن إما متغيرات عامة PUBLIC ، أو خاصة PRIVATE. وهذا مماثل تماماً لإعلان هذه المتغيرات على أنها متغيرات ذاكرة MEMVAR. وبما أنك ترغب الابتعاد عن هذه الإعلانات ، فستكون الحاجة لاستعمال هذا الخيار قليلة نسبياً.

خيار W/ Enable Warning (شغّل التحذير)

يرشد هذا الخيار المجمع إلى ضرورة إصدار "رسائل تحذيرية" لإسنادات المتغيرات غير المعلنة "أو المحيرّة". ونقترح استخدام هذا الخيار في كل مرة تريد فيها تجميع شيفر مصدر كليبر 5.x ولدى القيام بهذا سيصدر المجمع إنذاراً في كل مرة تنسى فيها إعلان متغير ما بحيث يضطرك هذا إلى كتابة شيفرة نظيفة.

وستكون هذه التحذيرات مزعجة أول الأمـر ، وخاصة عندما تعد بالمئات (أو بالآلاف أحياناً). وقد تضطر لتوجيه هذه التحذيرات إلى ملف نصوص خاص بها لكثرتها. بل قد تصبح بعض هذه الملفات أكبر من شيفرة المصدر الأصلية أحياناً. إلا أن هـذه الطريقة هي أفضل الطرق للتمكن مـن استخدام البرنـامج بشـكل سليم. وعندما تصبح قادراً على التجميع "بهدوء" (أي باستخدام خيار w/ دون ظهور أي تحذير) عندئذ فقط تعرف أنك أصبحت قادراً على كتابة برامج صحيحة باستخدام كليبر 5.x.

وهناك فائدة أخرى من استخدام خيـار w/ وهي أن يحذرك المجمع عن الأخطاء الطباعية التي ارتكبتها أثناء الكتابة. ويفيدك هذا في تصحيح الأخطاء الطباعية المزعجة قبل أن تقوم بعملية الربط وتنفيذ دورة البرنـامج. ويبين لك المثال التالي القصير الحالة التي يمكنّك خيار W/ من تجنُّبها :

```
function test
local lastname
// 50 lines of code ...
return lame // will generate a compiler warning for "LNAME"
```

الخيار x/ إنتاج قائمة من الإرشادات والاستدراكات

يوجه هـذا الخيـار الغـامض كليـبر 5.x المجمِّـع إلى إنتـاج قائمـة مـن الإرشـادات والاستدراكات. ويمكن الاستغناء عن هذا الخيار في المستقبل أو إبقاؤه ، إلا أنه يعطي حالياً نظرة جيدة على شكل تركيب ملفات الهدف التي تتعامل معها.

الخيار y/ يوقف التحسين

يوقف هذا الخيار (غير الموثق) المجمع عن مختلف أشكال عمليات رفع المستوى إلى الحد الأقصى (ماعدا " الاختصارات المنطقية "، التي يمكن إيقاف عملها باستخدام خيار Z/). وأكثر هذه الأمور وضوحاً هو "طي الثوابت constant folding" فإذا كانت لديك عبارة على الشكل التالي:

```
X := 1 + 2 + 3 + 4
```

ويقوم المجمِّع بتجميع هذه الثوابت وينتج الشيفرة في الهدف تمثل مايلي :

```
X := 10
```

لاحظ أنك إذا استخدمت الخيار P/ ، ثم فحصت ملف **PPO**. لن ترى أثراً لهذا ولانعتقد أن هناك كثيراً من المبرمجين يعرفون كل مايمكن أن تقوم به عملية التحسين إلى الحد الأقصى باستخدام كليبر 5.x. انتبه للاختبار التالي :

```
function main
local i , x
x := seconds( )
for  i := 1  to  10000
next
?  seconds( ) -  x
inkey (0)
x :=  seconds ( )
for i :=  1  to  9999
next
?  seconds( ) -   x
return nil
```

إن أول حلقة For..Next تستغرق مابين ١-٢ ثانية على جهاز **386sx / 20**، وأما الحلقة الثانية منها فلا تستغرق أي زمن على الإطلاق. وقد يبدو هذا الأمر غريباً ومدهشاً وغير مصدق. إلا أنه يمكن البرهنة على ذلك بإعادة التجميع باستخدام y/ ، والذي يمكن أن يستغرق فيه تجميع الحلقتين زمناً متساوياً.

خيار z/ أوقف عمل الاختصارات المنطقية

هذه الاختصارات أي: المنطقية هي النوع الثاني من ثلاثة أنواع لرفع مستوى الأداء إلى الحد الأقصى، والمبيته داخلياً في مجمّع كليبر 5.x. وتقوم هذه الاختصارات بدورها أثناء استخدام العوامل البولينية boolean مثل: (.AND. و .OR.). وهذا على خلاف كليبر Summer ' 87 حيث تجد السلوك المفترض لكليبر 5.x هو إيقاف تقييم العبارات الشرطية في عبارة ما عندما تكون العبارة التالية موضع نقاش. انتبه للمثال التالي:

```
lflag  :=   . t .
if   lflag  .or.   myfunc( )
    ?  "Made it this for "
endif
```

فبما أن LFLAG هو حقيقي (.T.) فسيعرف كليبر 5.x أن تقييم العبارة الشرطية IF كلها على أنها حقيقية. ولن يزعجك أبداً تقييم العبارة الشرطية الثانية في العبارة IF (والمسمى ( )MyFunc).

إن هذا الحديث صحيح لدى استخدام العبارة .AND.، مثال:

```
lflag  :=  .f.
if   lflag  .and.  myfunc( )
    ?  "Made it this for "
endif
```

مرة أخرى ، لن يتم تقييم ( )myFunc ثانية لأن LFLAG تفحص الخطأ (.F.) وهذا يعني أن كليبر 5.x سيدرك أن العبارة الشرطية IF يجب تقييمها كلها على أنها غير صحيحة ، وهكذا فلن يأخذ باعتباره العبارة الثانية (أو غيرها من العبارات الفرعية).

ولاشك أنك ستسعد بمجرى الحوادث بهذا الشكل ، إذ أن هذا يعني أن بمقدورك الآن دمج التعابير المختلفة في العبارة الشرطية IF ذاتها دون أن تخشى أن يحدث هناك عدم تطابق من أي نوع من الألوا ع. فمثلاً ، تعتبر الشيفرة التالي "كارثة" في كليبر 'Summer 87 إلا أنه يشتغل دون حدوث أي خطأ في كليبر 5.x.

```
if type( "mvar" )  == "C"  .and. mvar ==  "AQUARIUM"
   ? mvar
endif
```

وكذلك ، فإنه في كليبر 87 "Summer ، إذا حدث (mvar) على أنه كان أي شيء آخر عدا كونه مصفوفة حرفية ، فسيتحطم البرنامج ويتوقف بسبب خطأ عدم المطابقة إذ أن كليبر سيجبر على المتابعة وتقييم العبارة الثالية.

ومع ذلك ، فقد اعتمد بعض المبرمجين على عدم وجود هذه الاختصارات في برنامج Summer"87 ليقوموا بسلسلة من الأحداث المختلفة . فمثلاً : يمكنهم أن يعتمدوا على الوظيفة ( )MyFunc بحيث يمكن تقييمها دائماً في العبارة التالية :

```
if clause1  .and.  clause2 .and.  myfunc( )
```

فإذا كان لديك كليبر Summer"87 الذي يعتمد على هذا السلوك ، فاستخدم خيار المجمع Z/ لتجاوز الاختصارات المنطقية. ويرجى ملاحظة مايلي : إن أي برنامج يتم تنفيذه باستخدام عامل ماكرو (&) يستخدم الاختصارات دائماً. ولن يكون لخيار Z/ أي أثر في فعل هذه الحالة على الإطلاق.

مسح الشيفرة الميتة Dead code

مما تقدم تلاحظ أن كلاً من خياري y/ و z/ تؤثر على اثنتين من طرق تحسين مستوى الأداء إلى الحد الأقصى optimization. وأما الثالثة وهي "مسح الشيفرة الميتة" فتحتاج إلى مزيد من النقاش والتفصيل. فالشيفرات الميتة هي الشيفرات التي لايمكن تنفيذها. حاول مثلاً تجميع المثال التالي:

```
function main
if   .f.
     MyFunc( )
endif
return  nil
```

فإذا استخدمت الخيار p/ لتجهيز ملف مخرجات للمعالج الأولي ، فإنك سوف ترى استدعاء الوظيفة ( )MyFunc لايزال هناك. إلا أنك إذا فحصت الهدف ، فإنك لن ترى أية إشارة فيه على الإطلاق لهذه الوظيفة ( )MyFunc. وذلك لأن المجمّع يدرك أن الكتلة الشرطية IF لايمكن تنفيذها على الإطلاق نظراً لوجود خطأ (.F.) ثابت فيها. (وينطبق هذا الأمر على الثوابت فقط ، إذ أن المجمع لايستطيع أن يحكم على صحة المتغيرات من خطئها).

ويعتبر مسح البرامج الميتة مفيداً لأنه يصغّر ملفات الهدف ، وبالتالي يصغّر الملفات التنفيذية. إلا أنه مع ذلك ، قد يسبب شيئاً من الإزعاج عند تجهيز موجهات المعالج الأولي. وسنبين لك مزيداً من التفاصيل عن هذا الموضوع عند الحديث عن المعالج الأولي.

الإعدادت المقترحه

نقترح استخدام خيارات المجمع التالية بشكل دائم:

ليس هناك إجراء بدء ضمني	/n
إصدار تحذيرات	/w

وفّر شيئاً من الوقت والجهد على نفسك باستخدام متغير البيئة CLIPPERCMD (انظر القسم التالي أدناه) بحيث تجهز هذه الخيارات على أنها خيارات مفترضة لمجمّعك بدلاً من كتابتها في كل مرة تريد أن تتعامل معها.

التعامل مع الملفات

يتطلب مجمّع كليبر 5.x ٢٥ ملفاً على الأقل. تأكد من أن عدد الملفات الذي يشتمل عليه ملف Config.sys هو ٢٥ ملفاً على الأقل. أما إذا كنت تعمل على شبكة

نوفيـــل فيجـــب أن تفعـــل الأمـــر ذاتـــه في عبـــارة FILE HANDLES في ملـــف .SHELL.CFG



بيئة المجمِّع COMPILER ENVIRONMENT

هناك ثلاثة متغيرات بيئية تؤثر على مجمّع كليبر 5.x وهي:

متغير البيئة SET CLIPPERCMD

يمكنك هذا المتغير من توفير كثر من الوقت إذ يمكنك من تأسيس خيارات مجمّع كليبر 5.x. وفي كل مرة تجمّع فيها ملف شيفرة المصدر source code ، سيتم إلحاق أي شيء يحتويه متغير البيئة CLIPPERCMD للأمر.

فمثلاً : إذا احتوى متغير البيئة CLIPPERCMD على "/w /n" فإن أمر التجميع سيكون كالتالي:

```
Clipper myprog
```

فسيتم معاملتها وكأنك كتبت مايلي:

```
Clipper myprog   /n  /w
```

ويضبط هذا المثال التشكيل المفضل للمجمّع بالطريقة التي ذكرناها أعلاه .

```
SET CLIPPERCMD= / n   / w
```

إذا كنت تحاول اكتشاف الأخطاء البرمجية في البرنامج الذى أعددته ، فيجب أن تأخذ بعين الاعتبار إضافة الخيارين التاليين /p و /b إلى CLIPPERCMD.

ويمكنك أن تمضي في متابعة تحديد خيارات سطر الأوامر ، علاوة على تلك التي تم تحديدها باستخدام CLIPPERCMD. ولابد أن تنتبه إلى بعض التصرفات الغريبة التي قد تنتج عن إستعراض بعض الخيارات والتعامل معها.

إن خيار d/ (حدِّدْ معرِّفاً للمعالج الأولي): يمكنك من تحديد أكثر من معرِّف واحد على سطر الأوامر ، حتى وإن كنت حدّدت معرِّفاً واحداً أو أكثر باستخدام CLIPPERCMD. إلا أنك إذا أعدت تحديد معرِّف ما ، فسيصدر لك المجمع رسالة تحذيرية مناسبة.

أما الخيار i/ (ضمِّن مسار البحث عن ملف) يمكنك من تحديد أي عدد تريده من أمر INCLUDE حسب اللزوم ، علاوة على تلك الأوامر الموجودة في ملف CLIPPERCMD.

أما الخيار o/ (مكان ملف الهدف) إذا حددت هذا الخيار على سطر الأوامر ، فإنه سيتجاوز أي تجهيز مسبق في متغير البيئة CLIPPERCMD.

أما الخيار p/ (أنتج ملف PPO) إذا كنت قد حددت هذا الأمر في متغير البيئة CLIPPERCMD ، فإن إضافته إلى سطر الأوامر سيوقف تشغيل هذه الميزة.

أما الخيار r/ (بحث المكتبة) يمكنك من تحديد أي عدد من المكتبات الإضافية التي تريد أن تبحث فيها حسب الضرورة ، وذلك علاوة على تلك المكتبات المذكورة في CLIPPERCMD.

أما الخيار t/ (مكان الملفات المؤقتة) إذا أضفت هذا الخيار إلى سطر الأوامر ، فإنه سيتجاوز أي تجهيز سابق لأمر t/ كنت قد وضعته سابقاً في المتغير CLIPPERCMD.

والخيار u/ (ملف الترويسة القياسي الذي يجب استخدامه) إذا حددت هذا الخيار على سطر الأوامر ، فإنه سيتجاوز أي تجهيز سابق للأمر ذاته في متغير البيئة CLIPPERCMD (إن وجد).

ضبط موجه التضمين INCLUDE

يوجه هذا الخيار INCLUDE المجمّع أين يبحث عن ملفات الترويسة (.CH) أما ترتيب البحث فهو:

١- الدليل الحالي.

٢- أية ملفات تم تحديدها باستخدام خيار المجمع i/.

٣- أية ملفات تم تحديدها باستخدام متغير البيئة INCLUDE.

ويوجه الأمر التالي ، مثلاً ، المجمع للبحث عن ملفات الترويسة في الدليل C:\CLIPPER5\INCLUDE ، إذا لم تكن موجودة في مكان آخر غير المكان الحالي وهو:

```
SET   INCLUDE=  C:\ clipper5 \ include
```

ضبط الخيار SET TMP

يوجه متغير TMP المجمّع والرابط إلى المكان الذي يعدان فيه الملفات المؤقتة. وكما هو الحال في خيار المجمع t/ فإن هذا الخيار هو أفضل مايكون للذاكرة العشوائية للقرص ، إذا كانت متوفرة لديك. ويوجه المثال التالي الملفات المؤقتة إلى القرص :D.

```
SET  TMP = D:\
```

تحذير

وكما هو الحال في خيار المجمع t/ ، فإنك إذا استخدمت متغير البيئة **TMP** ، لابد من أن تتأكد أن السواقة / المسار **drive\ path** تشير إلى مسار موجود فعلاً ، وإلى مساحة كافية متوفرة فعلاً على القرص لاحتواء الملفات الكبيرة للبرامج **PRG.** التي تريد تجميعها وإلا فستصدر رسالة خطأ مزعجة تقول : "لايمكن تجهيز ملف وسيط "Cannot create intermediate file" أثناء القيام بعملية التجميع.

الخيار SET CLIPPER Parameters

إن معظم هذه المتغيرات parameters تدور حول كليبر 5.x الذي يحتوي على برنامج "نظام إدارة الذاكرة التخيلية" Virtual Memory Management System. ولاحاجة لك لاستخدامها فوراً ، ولكن يستحسن أن تلم بها وتتعرف عليها.

الخيار BADCACHE

يمكن استخدام هذا الخيار عند ملاحظة تضارب مع برامج أخرى تستخدم EMS (مثل البرامج المقيمة في الذاكرة TSR أو disk caches). حيث يجبر هذا الخيار برنامج الذاكرة التخيلية أن يحفظ حالة صفحة EMS ، ويعيده إلى وضعه الأصلي خلال كل مرة يتم فيها التوصل إلى برنامج EMS. ويجب أن نحذرك هنا إلى أنه في بعض البرامج قد يؤثر هذا الخيار BADCACHE على أداء برامج الذاكرة التخيلية.

الخيار CGACURS

يستثني هذا الخيار استخدام قدرة المؤشر الموسع لكل من الكروت طرازي EGA/VGA . ويجب استخدامه فقط عندما تلاحظ تصرفاً غير عادي للمؤشر على الشاشة أثناء تشغيل برامجك في بيئة متعددة المهام multi-tasking ، أو في بيئة البرامج المقيمة في الذاكرة TSR.

الخيار DYNF

يعرف هذا الخيار الحد الأقصى لعدد الملفات (File handles) ، المسموح بها في نظام الإحلال الديناميكي لـ: RTLINK. ويجب أن يكون هذا الرقم ضمن المدى من ١–٨. وإذا لم يتم تحديد هذا التجهيز ، فسيكون المفترض هو ٢ فقط.

الخيار INFO

لدى تحديد هذا التجهيز ، سيتم عرض معلومات محددة عند بدء التشغيل والخروج من البرنامج. وتتضمن هذه المعلومات كلاً من: رقم إصدار كليبر المستخدم لتجميع برنامجك و نقطة بداية دخول البرنامج ، حجم الذاكرة الاعتيادية والموسعة المتوفرة على القرص. ونبين فيما يلي عينة عن الإخراج الذي يصدر عن استخدام خيار INFO (لاحظ أن عبارة "Hello world" هي الإخراج الحقيقي الوحيد الذي يصدر عن البرنامج.

```
D: \ > test
Clipper  (R) 5.2a ( rev  197 ) ASCII
DS =4FE1 :0000  DS  avail =36KB   OS  avail =214KB   EMM  avail =0KB
      hello world
 (Fixed heap =13KB / 1)
D: \ >
```

الخيار NOALERT

يسبب هذا الخيار أموراً عجيبة لدى استخدامه في كليبر 5.x. كما أن برنامجه غير محدد ويمكن أن تداعب زملاءك الذين يستخدمون الوظيفة ( )ALERT داخل برامجهم، بوضع الخيار NOALERT في متغير البيئة الخاص بهم. وستكون نتيجة برامجهم غامضة تماماً وعلى شكل ألغاز ، وسيصدر وقت تشغيل خاطيء من السطر الذي يتم فيه استدعاء الوظيفة ( ) ALERT.

الخيار NOIDLE

إن كثيراً من برامج كليبر 5.x تحتوي على كثير من "الوقت الميت" (أي الوقت الذي ينتظر فيه البرنامج المستخدم لاختيار خيار ما من القائمة الرئيسه، أو إدخال بيانات ما). ويكتشف كليبر هذا الوقت الميت أثناء تنفيذ البرنامج ، ويستغله للتخلص من بعض النفايات والقيام ببعض الأعمال الداخلية للتنظيف الذاتي. ويحسن هذا الاستغلال الفعال

للوقت الأداء العام للبرنامج. وإذا لم ترد أن يقوم البرنامج بإستغلال هذا الوقت الميت ، فيمكن استخدام الخيار NOIDLE ، لإيقافه عن العمل. إلا أننا نقترح عدم استخدام هذا الخيار لأنه يقلل من أداء النظام.

الخيار SQUAWK

يزودك هذا الخيار العجيب بتغذية إرجاعية صوتية أثناء تشغيل برنامجك. إلا أن دقته لم تؤكد بعد ، غير أنه يبدو متعلقاً بعمل إدارة الذاكرة الخيالية "VMM" Virtual memory manager. ويمكن أن تستعمل هذا الخيار لتتعرف على الأماكن التي توجد فيها بعض المشاكل في برنامجك (مثل كثير الأنشطة المتعلقة بإدارة الذاكرة الخيالية). جمّع البرنامج الاختياري التالي ، وشغّله لاستعراض كيفية عمل هذا الخيار باستخدام // SQUAWK. (ملاحظة : إن كافة النغمات التي يصدرها الجهاز باستخدام هذا الخيار هي ذات صوت واحد مستمر).

```
funtion main
Local  a  : = { }
do    while   inkey ( )   ==   0
   aadd ( a , space(1000))
   @  1, 1 say  len (a)
enddo
return  nil
```

الخيار SWAPK

يحدد هذا الخيار الحجم الأقصى للملف الذاكرة الخيالية التبادلية VM بالكيلوبايت. ويمكنك تحديد أي رقم تشاء بدءاً من ٢٥٦ إلى ٦٥٥٣٥. أما إذا لم تحدد هذا التجهيز فإن كليبر سيستخدم التجهيز المفترض وهو ١٦٣٨٤ (١٦ميجابايت). ويرجى الانتباه أيضاً إلى أن تحديد حجم هذا الملف التبادلي ، لايعني بحال من الأحوال أن حجم الملف فعلياً سيصل إلى هذا الرقم الكبير. كما يجدر إنذار المبرمجين هنا إلى إيقاف عمل التبادل مع القرص (Disk

swappin\g) فإنه قد يتسبب في فشل البرنامج وإصدار رسالة الخطأ المقيتة والمزعجة: ("Out of Memory" لاتتوفر مساحة كافية في الذاكرة).

الخيار SWAPPATH

يحدد هذا الخيار المسار أو الدليل الذي ستكتب فيه الملفات التبادلية للذاكرة الخيالية VM. وإذا لم تحدد هذا التجهيز ، فإن الملف التبادلي هذا سيكتب في الدليل الحالي. وإذا كان لديك حجم كبير من الذاكرة العشوائية للقرص RAM DISK فيستحسن استخدامه كمسار توجه إليه الملفات التبادلية. ويجب تحديد المسار ضمن علامات اقتباس "".

يمكن أن تستفيد برامج الشبكات أيضاً من هذا الخيار. ويمكن أن يكون أحد الأمثلة هو تجهيز خيار SWAPPATH لتجهيز ملف تبادلي على محطات العمل في القرص الصلب لتقليل أنشطة الشبكة.

أما السبب الوجيه الثاني فهو عندما لايكون للمستخدم حق إنشاء ملفات في دليل الشبكة الذي يوجد فيه التطبيق (الملف التنفيذي). فمن الأهمية بمكان أن يكون نظام الذاكرة الخيالية VM قادراً على إنشاء ملف تبادلي بناء على الحاجة ، بحيث لايتحطم البرنامج إذا لم يكن لدى المستخدم امتياز "الكتابة" لذلك ، يجب تجهيز SWAPPATH لتبادل الدليل الشخصي للمستخدم. ويمكن تنفيذ هذا عادة بتجهيز ملف مجموعة أوامر batch file على النحو التالي:

```
ExeName // SWAPPATH : "f : \ users \ %USERID%"
```

حيث أن USERID متغير بيئة مجهز في ملف تشغيل الشبكة للمستخدمين.

الخيار TEMPPATH

يتحكم هذا الخيار بوضع الملفات المؤقتة التي تم إنشاؤها خلال عمليات الفهرسة والفرز مايسمى (... gasp...). ويجب تحديد المسار path ضمن علامات تنصيص. فإذا استخدمت هذا الخيار ، يجب أن تتذكر أن الملفات المؤقتة الناتجة عن الفهرسة والفرز قد تكون كبيرة الحجم نسبياً. لذلك ، فقد يكون تجهيز المسار المؤقت في منطقة صغيرة سبباً لإخفاق عمليات الفهرسة والفرز. فيستحسن أن يكون هناك حجم المساحة الفارغة في الذاكرة يعادل على الأقل ضعف أكبر ملف فهرسة أو ملف قاعدة بيانات المطلوب فرزه.

ويمكن استخدام خيار TEMPPATH مثل سابقة SWAPPTH لخفض كمية الأنشطة على الشبكة.

مثال

يستخدم المثال التالي بعض الخيارات التي تشكل بيئة "وقت التشغيل".

```
SET CLIPPER=DYNF:4;SWAPK:4096;SWAPPTH: " e:\";CGACURS;INFO
```

لاحظ أن النقطتين " : " بين كل كلمتين والتجهيز المطابق لهما هو أمر اختياري. فعلى سبيل المثال: فإن **"DYNF:4"** ستعمل مثل **"DYNF4"** تماماً. كما يمكنك أن تسبق كلاً من هذه الخيارات بشرطتين مائلتين ("//").

إعداد سطر الأوامر

يتيح لك كليبر 5.x تحديد تجهيزات بيئة وقت التشغيل على سطر الأوامر إذا رغبت ذلك. ولابد أن تضع شرطتين مائلتين (" // ") قبل كل تجهيز منها ، كما يجب أن تلاحظ أنه لابد أن تسبق هذه المتغيرات أي متغيرات عادية تضعها في برنامجك. فإذا نويت استخدام

هذه الميزة في برنامجك فيستحسن إعداد ملف مجموعة أوامر batch file يحتوي علـى تلـك المتغيرات بحيث لا يحتاج مستخدم البرنامج أن يتعامل معه.

ويبين لك المثال التالي كيف يمكنك استخدام سطر الأوامر لتحديد التجهيزات الـتي تم الحديث عنها أعلاه:

```
D:>myapp / /DYNF: 4 / /SWAPK:4096 / /SWAPPATH:"e:\ " / /CGACURS / /INFO
```



برنامج كشف الأخطاء DEBUGGER

غالباً ما يتم تجاهل هذا البرنامج ، علماً بأنه إحدى الإضافات الهامة والمتميزة في كليبر 5.x. ولهذا ، فإن هذا البرنامج بالذات يجعل كليبر 5.x لغة برمجة حقيقية. ويمكن القول باختصار إن برنامج "اكتشاف الخطأ وتصحيحه" Debugger هو قفزة نوعية رائعة تميز هذا البرنامج عن غيره من البرامج السابقة مثل كليبر **Summre' 87**. ونهدف من هذا النقاش إلى أمرين هامين. (أ) أن نعرِّف المستخدم بميزة اكتشاف الخطأ وتصحيحه. (ب) تحسين قدرتك لاستخدام هذه الميزة بإعطائك كثيراً من الإرشادات والملاحظات والطرق المختصرة التي اكتشفها مؤلف الكتاب أثناء تعامله مع هذه اللغة خلال السنتين الماضيتين.

إعداد شيفرة المصدر الخاصة بك

إذا رغبت بمشاهدة شيفرة المصدر والانتقال فيها خطوة خطوة في كليبر باستخدام برنامج اكتشاف الخطأ وتصحيحه ، فيجب أن تجمّع برنامجك باستخدام الخيار **B/** ، كما يجب أن تبقي أرقام السطور ، أي : لا تستخدم الخيار **L/** .

البدء بتشغيل البرنامج Debugger

إن تشغيل هذا البرنامج سهل جداً ، اكتب "CLD" يعقبه اسم البرنامج الذي تريد اكتشاف الخطأ فيه فقط. فإذا أردت تمرير متغيرات سطر الأوامر إلى برنامجك ، فيجب أن تعقب برنامجك بهذه الأوامر.

متغيرات سطر الأوامر

الخيار CLD /S <appName> <appParams>

إن خيار s/ يشغل وضعية أو طور تقسيم الشاشة (EGA/VGA) فقط. وستظهر شيفرة المصدر في الجزء السفلي من الشاشة ، ويمكنك في هذه الحالة مشاهدة مخرجات برنامجك في الجزء الأعلى من الشاشة. وإذا كانت الشاشة التي عندك هي أحد نوعي EGA / VGA فلعل هذه الطريقة هي أكثر الطرق فاعلية لاستخدام برنامج Debugger.

الخيار CLD /43 <appName> <appParams>

يشغل هذا الخيار وضعية أو طور عرض ٤٣ سطراً على الشاشة (EGA/VGA) ، وستظهر شيفرة المصدر على الشاشة بأكملها.

خيار CLD / 50 <appName> <appParams>

يشغل هذا الخيار وضعية عرض ٥٠ سطراً على شاشة (VGA) وستظهر شيفرة المصدر على النافذة بأكملها.

الخيار CLD @<scriptfile> <appName> <appParams>

يمكنك هذا الخيار من تحميل خيارات من ملف مكتوب (script file) ومخزون سابقاً. وسنتكلم بعد قليل عن ملف Script File بمزيد من التفصيل.

التعامل مع قوائم الاختيارات

اضغط على مفتاح Alt ، وأبق أصبعك ضاغطة عليه ، واضغط في نفس الوقت ذاته على أول حرف من قائمة الاختيارات التي تريد التعامل معها ، إذا أردت التعامل مع أي قائمة اختيارات من قوائم اختيارات برنامج اكتشاف الأخطاء (Debugger). فعلى سبيل المثال: إذا أردت التعامل مع قائمة اختيارات الملف ، اضغط على مفتاح F-Alt واضغط مفتاح F في الوقت ذاته ، وعند انسدال القائمة أمامك على الشاشة ، يمكنك اختيار أي خيار منها ، إما باستخدام أسهم الاتجاه ثم الضغط على مفتاح Enter ، أو بكتابة الحرف المطابق للخيار المطلوب. فمثلاً : إذا أراد المستخدم فتح نافذة عرض Monitor Window للخيارات LOCAL ، فيمكن الضغط على مفتاح M-Alt، ثم الضغط على حرف L.

لاحظ أن كثيراً من خيارات قائمة الاختيارات تتميز بأن لها مفاتيح سريعة يمكن تنفيذ الأوامر باستخدامها. ويستحسن أن تحاول تعلم كيفية استخدام هذه المفاتيح السريعة بدلاً من استخدام قوائم الاختيارات ، أو يجب أن تتحمل إضاعة كثير من الوقت أثناء التعامل مع البرنامج بسبب ذلك. كما تجدر الإشارة إلى أنه تم إدراج كافة المفاتيح المعنية بأعمال محددة في برنامج اكتشاف الأخطاء وتصحيحها على السطر الأسفل ، وذلك للمراجعة السريعة.

قائمة اختيارات "ملف" File

تحتوي هذه القائمة على ثلاثة خيارات وهي :

الفتح (**Open**): يفتح أي ملف نصوص لمشاهدته. ويعتبر هذا الأمر مفيداً من جهتين.

(أ) فتح ملفات PRG أخرى من برنامجك لتجهيز نقاط محددة فيها ، و (ب) الإشارة السريعة إلى الملفات المتعلقة ببرنامجك ، كملفات الترويسة. لاحظ أنه لايمكنك مشاهدة أكثر من ملف واحد في وقت واحد فقط ، ولابد من إغلاق هذا الملف لمتابعة عملية اكتشاف الأخطاء وتصحيحها في الملفات الأخرى.

تابع (Resume): يغلق الملف الذي تم فتحه لمراجعته ومشاهدته ، ثم يتابع عملية اكتشاف الأخطاء وتصحيحها في تلك الجلسة. لاحظ أن الضغط على مفتاح F5 (تشغيل Rum) أو الضغط على مفتاح F8 (خطوة واحدة Single step) لمتابعة تنفيذ برنامج ما سيكون له الأثر ذاته.

التعامل مع دوس (DOS Access): يمكنك هذا الخيار من الخروج من البرنامج والتعامل مع دوس. كما أن هذا الخيار يفحص "المسار" بحثاً عن نسخة من مفسِّر الأوامر (COMMAND.COM) ، وإذا وجد الملف ، يحمِّله بحيث يكون لديك برنامج غلاف دوس مؤقت. إلا أنه لا يفحص المتغير البيني الذي يسمى COMSPEC (والذي يكون أكثر فائدة ومعنى). لذلك ، إذا أبقيت مفسِّر الأوامر في دليل غير محدد لمسارك ، فإنك سترى رسالة مفادها: "Type 'exit' to return to the Clipper debugger" . اكتب **exit** للعودة إلى برنامج كشف الأخطاء Debugger ، وستعود فوراً إلى ذلك البرنامج.

إلا أنني أشعر أن تسمية هذه العملية " نظرة سريعة على دوس " أفضل من الاسم الحالي لها. (لاحظ أن اسم ملف COMMAND.COM قد تم تشفيرها بشكل واضح في برنامج **Debugger** ويعني هذا أن أية مفسّرات أو أوامر بديلة مثل **4DOS** سيتم تجاهلها تماماً).

خروج **(eXit)**: (المفتاح السريع هو : ("Alt-X")) وهذا واضح من عمل هذا المفتاح. كما يمكنك الخروج من البرنامج أيضاً بكتابة كلمة "Quit" أو حرف "Q" في نافذة الأوامر.

قائمة اختيارات "ابحث عن مكان" (Locate)

تحتوي هذه القائمة على خمسة خيارات جميعها واضحة ، ومع ذلك فسنبينها هنا لتمام الإيضاح:

Find (ابحث عن): ابحث عن أول مكان تظهر فيه سلسلة نصية (حرفية).

Next (التالي): ابحث عن المكان التالي الذي تظهر فيه السلسلة الحرفية ذاتها.

Previous (السابق): ابحث عن المكان السابق الذي تظهر فيه السلسلة الحرفية ذاتها.

Goto Line (اذهب إلى سطر): يقفز إلى سطر محدد. وقد يكون هذا الخيار هو أكثر الخيارات استخداماً في هذه القائمة.

حسّاس للحالة (**Case Sensitive**): يشغل هذا الخيار ، ويوقف عمل البحث ، ليصبح حساساً للحالة أو غير حساس لها.

ملاحظة
خيارات البحث هذه تنطبق على ملف PRG فقط ، أما إذا كنت تراجع ملف شيفرة المعالج الأولي ، فلا تتوقع أن تستطيع البحث عن أي شيء فيه.

قائمة اختيارات (شاهد) View

تحتوي هذه القائمة على أربعة خيارات وجميعها مفيدة ، وهي:

جهز (**Sets**): يمكنك هذا الخيار من مشاهدة كافة تجهيزات النظام العامة للبرنامج. كما يمكنك من تغييرها بسرعة إذا أردت On-the-fly. إلا أنه يجب الانتباه إلى أنك إذا غيرت أياً منها ثم أردت أن تفعل بها شيئاً آخر بعد ذلك في البرنامج ، فإن البرنامج سيتجاهل أي تغيير تم من خلال برنامج اكتشاف الأخطاء وتصحيحها (**Debugger**).

مناطق العمل (Workareas) (المفتاح السريع: F6)

يمكنك هذا الخيار من مشاهدة المعلومات المتوفرة عن كافة مناطق العمل المفتوحة. وتتضمن هذه المعلومات كلاً من:

- Alias الاسم المستعار.
- منطقة العمل النشطة.
- الوضعية الحالية لمؤشر السجل.
- عدد السجلات الإجمالي.
- الحالة الراهنة لبداية الملف ، **BOF( )** ونهاية الملف **EOF( )**.
- شرط الترشيح Filter condition.
- كافة مفاتيح الفهرسة في منطقة العمل هذه.
- بنية قاعدة البيانات.
- محتويات السجل الحالي.
- العلاقات التي تحتوي عليها منطقة العمل هذه.
- السواقة الحالية لقاعدة البيانات في منطقة العمل هذه (الإصدار 5.2 فقط).

يمكنك التنقل ما بين النوافذ باستخدام مفتاح الجدولة الأمامية [Tab]. لاحظ أنك إذا كنت في النافذة الوسطى ، يمكنك إخفاء أي شيء أو تكبيره في منطقة العمل. وإذا أردت أن تفعل ذلك ، فما عليك إلا أن تعلّم الشيء الذي تريد إخفاءه أو تكبيره ثم تضغط على مفتاح [Enter]. وتتضمن الأمثلة على هذه الأمور كلاً من : "معلومات منطقة العمل" و "السجل الحالي".

لاحظ أنه إذا احتوت قاعدة البيانات على حقول أكثر مما تتسع له الشاشة فيمكنك التنقل إلى أسفل خلال هذه الحقول لمشاهدة القيم الحالية فيها.

خيار "شاشة البرنامج" App Screen (المفتاح السريع: F4):

يمكنك هذا الخيار من مشاهدة شاشة البرنامج الحالي. اضغط على أي مفتاح على لوحة المفاتيح للعودة إلى برنامج اكتشاف الأخطاء وتصحيحها **Debugger** .

الخيار Callstack

يمكنك هذا الخيار من تشغيل نافذة **Callstack** وإيقافها عن العمل ، والتي تظهر بشكل افتراضي طبيعي على الجانب الأيمن للشاشة العادية. وتبين نافذة **Callstack** اسماء كافة الأعمال فيها. فإذا انتقلت إلى تلك النافذة ، وحرّكت السهم إلى أعلى أو إلى أسفل فيها فإن نافذة البرنامج ستنتقل إلى السطر المحدد حيث كنت في كل عمل من الأعمال. وهذه الميزة رائعة جداً وتوفر لك قدراً كبيراً من الوقت. لاحظ أنه لايهم وجود هذه الوظائف في ملفات برامج PRG منفصلة ، فإن برنامج **Debugger** سيقفز إلى الوظيفة المطلوب فوراً.

قائمة اختيارات "التنفيذ" RUN Menu

تشتمل معظم خيارات هذه القائمة على مفاتيح تنفيذ سريعة ، والتي نقترح تعلمها واستخدامها لتسهيل القيام بمختلف الأعمال بسرعة. إلا أننا نرى أن هذه الخيارات تستحق الشرح والتوضيح.

أعد التشغيل (**Restart**): إذا توقف برنامجك عن العمل ، إما لسبب عادي أو غير عادي بسبب وجود خطأ فيه فلا بد من إعادة تشغيله مرة ثانية للتعامل معه. ويمكنك هذا الخيار من القيام بذلك.

البرنامج التالي (**Next routine**): تمت إضافة هذا البرنامج مع الإصدار **5.2** من كليبر. وإن اختيار هذا الخيار سيشغل البرنامج ثم يتوقف عند أول عبارة تنفيذية في الإجراء التالي أو الوظيفة الذي تواجهه. وهذا يؤثر في تقدم برنامج Debugger للوظيفة التالية. أما

المفتاح السريع لهذا الاختيار فهو F5-Ctrl. (ومن الطبيعي ، أن تكون قد قمت بتجميع الإجراء أو الوظيفة باستخدام المفتاح B/ لكي يتوقف برنامج Debugger داخله).

Animate: يمكنك هذا الخيار من تشغيل برنامجك وتضع فترة توقف قصيرة بين كل عبارة والتي تليها. ويتم ضبط درجة التوقف بخيار السرعة(انظر أسفل).

الخطوة Step (المفتاح السريع هو: F8): يمكنك هـذا الخيـار مـن الأنتقال خطوة خطوة على سطور برنامجك. وإذا استخدمت هـذا الخيـار بـدلاً مـن استخدام المفتـاح F8 فـأنت تبحث عن التعب والمشاكل.

الأثر (Trace) (المفتاح السريع هو F10) : لاشـك أن هـذه التسـمية هـي " جنحـة " إذ أنك تستخدم هذا المفتاح عندما لا تريد تتبع أثر وظيفة ما. فإذا كنت تتنقـل بـين خطـوات برنـامجك ، ووصلـت إلى وظيفـة لا تريـد تتبـع أثرهـا ، اضغـط على مفتـاح F10 فيتـم استدعاء هذه الوظيفة وتنفيذها دون حاجة لأن تمر بأية خطوة.

انطلق Go (المفتاح السريع هو: F5): ينفذ هذا الخيار برنامجك بكامل سرعته.

إلى المؤشر to Cursor (المفتاح السريع هو: F7): يقوم هذا المفتـاح بتنفيـذ التطبيـق حتـى تصل إلى سطر شيفرة مصدر معلّم بالمؤشر في نافذة شيفرة المصدر. ويُعتبر هذا الخيار عملياً جداً لضبط نقاط توقف مؤقت breakpoints.

تنبيـــه
إذا استخدمت هذا الخيار لإيقاف عبارة يتم تتابعها على عدة سـطور ، فيجـب التـأكد مـن وضع المؤشر على آخر سطر من سطور التتابع ، وإلا فلن يتوقف تنفيذ البرنامج.

السرعة sPeed: يضبط هذا الخيـار سـرعة الرسـوم المتحركـة animation طبقـاً لأعشـار الثانية ، والافتراض هنا أنه ليس هناك توقف لحظي.

قائمة الاختيارات النقطية Point Menu

تمكنك هذه القائمة من تحديد "نقاط مراقبة" و "نقاط تتبع أثر" و "نقاط توقف".

خيار "نقطة المراقبة" Watchpoint: يمكنك هذا الخيار من تحديد نقطة مراقبة ، ويمكن أن تتضمن نقطة المراقبة أي تعبير صحيح من تعبيرات كليبر ، بما في ذلك القيمة الراجعة من وظائف كليبر ، أو الوظائف المعرفة من قبل المستخدم. ولدى التنقل بين مختلف فقرات برنامجك. يتم تغيير قيم نقاط المراقبة هذه طبقاً للمرحلة التي أنت فيها في البرنامج.

خيار "نقطة تتبع الأثر" Tracepoint: تعتبر هذه النقاط مماثلة تماماً لنقاط المراقبة بإضافة رئيسة واحدة وهي: إيقاف لتنفيذ البرنامج لدى تغيّر هذه القيم. وإن أفضل الأمثلة على نقاط التتبع هي الوظائف التي يقوم بها كليبر مثل: ( )RECNO و ( )EOF. لاحظ أنه ليس من المستحسن استخدام متغيرات LOCAL كنقاط تتبع وذلك نظراً للطريقة التي يتم بها إيقاف عملها في كل مرة يتم فيها إدخال الوظيفة.

خيار "نقطة التوقف" Breakpoint (المفتاح السريع هو: F9) يشبه هذا الخيار إلى حد بعيد خيار "المؤشر" في قائمة اختيارات التشغيل ، فيما يخص قدرة إيقاف تنفيذ البرنامج عند الوصول إلى السطر الحالي للمؤشر. إلا أن هذا الخيار على خلاف "المؤشر" يبقى سارياً خلال فترة جلسة عمل اكتشاف الأخطاء وتصحيحها. إذا أردت تحديد "نقطة توقف" ، أنقل المؤشر إلى السطر المطلوب، ثم اضغط على مفتاح F9. لاحظ أن عملية "المؤشر" تنطبق على "نقطة التوقف" ، فإذا حاولت تحديد نقطة توقف على سطر يحتوي على تتابعات ، لابد أن تتأكد من وضع المؤشر في آخر سطر من سطور التتابع.

لاحظ أيضاً أن برنامج اكتشاف الأخطاء وتصحيحها Debugger لن يسمح لك بتحديد نقاط توقف على أية عبارة تبتدىء بأي من إعلانات LOCAL أو STATIC. وذلك أنه يعتبرهما عبارتين غير قابلتين للتنفيذ ، ويعتبر هذا الأمر بمجمله إفتراضاً سليما. ومع ذلك فإن عبارة LOCAL التي تعيّن قيمة وهي في الحقيقة تنفيذية ، وبالتالي فيجب أن تسمح بتحديد نقطة توقف. وإذا أردت تجاوز برنامج اكتشاف الأخطاء وتصحيحها العسير ،

فيمكنك كتابة مايلي: ##BP في نافذة الأمر ، حيث يمثل رمزاً ## رقم السطر الذي تريد فيه تحديد نقطة توقف. وإن تحديد نقطة توقف بهذه الطريقة يلغي عمل الفحص الداخلي الذي يقوم به البرنامج Debugger. (كما يمكن أيضاً تحديد نقاط توقف على أسطر لا وجود لها ، إلا أنني آمل أن تكون من الحكمة والأناة والفهم بحيث لا ترتكب مثل هذه الحماقة لما لها من محاذير سيئة).

خيار "احذف" Delete: يمكنك هذا الخيار من حذف عناصر من نافذة المراقبة. كما يرجى الانتباه إلى أن برنامج كشف الأخطاء لكليبر الإصدار 5.2 ، يمكنك من حذف أية عناصر في نافذة المراقبة بتعليمها والضغط على مفتاح [Delete] بمنتهى البساطة.

كما يمكنك تغيير محتويات أية متغيرات معروضة في نافذة المراقبة ، أو نافذة التحكم/المراقبة بتعليم العنصر المطلوب والضغط على مفتاح [Enter] ويرجى الانتباه إلى أن هذا لا ينطبق على كل كتل الشيفرة code blocks التي لا يمكن تعديلها.

كما يمكنك تفتيش محتويات المصفوفات array والأهداف object المعروضة في نوافذ المراقبة أو التحكم. علّم العنصر المطلوب واضغط على مفتاح [Enter] مرتين. وسيتم عرض أي من عناصر المصفوفة أو متغيرات الهدف الحالي. ويمكنك تغيير أي من هذه العناصر بتعليمه ثم الضغط على [Enter] ، ولعل هذه الميزة تجعل برنامج اكتشاف الأخطاء وتصحيحها أداة فعالة وقيّمة جداً لتعلم عن المصفوفات المتداخلة المعقدة.

قائمة اختيارات الشاشة Monitor Menu

تمكنك الخيارات الموجودة في هذه القائمة من مشاهدة المتغيرات المختلفة لمجال ما. ويمكن أيضاً إيقاف أي مجال وتشغيله. ويبين العنوان الموجود في أعلى نافذة الشاشة مرجعاً سريعاً عن أنواع المتغيرات التي تشاهدها حالياً. أما خيار الفرز فيمكنك من فرز هذه المتغيرات طبقاً لاسمائها فقط.

وقد أضيف خيار جديد في برنامج كشف الأخطاء لإصدار كليبر 5.2 ، وهو: "ALL" (جميع) وهو كما هو واضح من اسمه يعمل على تحديد جميع المتغيرات المشاهدة. كما أنه أيضاً قابل للتشغيل والإيقاف (on/off) كبقية أوامر قائمة اختيارات الشاشة.

مراقبة المحليات **Locals** ضمن كتل الشيفرة : إذا فتحت نافذة مراقبة لمشاهدة متغيرات محلية فيمكنك مشاهدتها داخل كتل الشيفرة عند تقييمها. وإن هذا الخيار ، والذي يليه يجعلان برنامج اكتشاف الأخطاء وتصحيحها لا غنى عنه لدراسة سلوك كتل الشيفرة هذه.

قائمة الخيارات Options Menu

تمكنك خيارات هذه القائمة من تفصيل برنامج كشف الأخطاء Debugger ، وتجهيزه بالطريقة التي تريدها.

خيار "شيفرة المعالج الأولي" Preprocessed Code: عند تشغيل هذا الخيار سيتم عرض مخرجات المعالج الأولي مع شيفرة المصدر الخاصة ببرنامجك. ويجب الانتباه إلى أنه يجب تجميع ملف البرنامج باستخدام الخيار P/ لإنشاء ملف PPO. أما إذا وضعت هذا الخيار في وضعية التشغيل ولم تجهز الملف المذكور PPO. فستعرض نافذة المصدر رسالة مفادها: أن الملف غير متوفر.

خيار "تحذير" (Warning): ليس برنامج كشف الأخطاء (Debugger) من الذكاء بدرجة تمكنه من اكتشاف إن كان ملف PPO. يطابق تماماً ملف PRG.. فلنفترض أنك جمّعت شيفرة المصدر لبرنامجك باستخدام الخيار P/ ، فإذا قمت بتغييرات على البرنامج بعد ذلك ، ثم أعدت تجميعه ثانية دون استخدام الخيار P/ ، ثم أردت مشاهدة شيفرة المعالج الأولي باستخدام البرنامج Debugger فإنك ستشاهد الملف الحالي للملف PRG. ولكن باستخدام الملف القديم PPO. والذي قد لايطابقه بشكل مناسب. لذلك فإذا رأيت خلافات واضحة بين البرنامج وملف PPO. فيجب الخروج من برنامج Debugger وتصحيحها مباشرة وإعادة التجميع من جديد باستخدام الخيار P/.

خيار أرقام السطور Line Numbers: يمكنك هذه الخيار من تشغيل وضع إظهار أرقام السطور أو إيقاف عمله ، إلا أنه لا يحذفها من البرنامج ، ولايؤثر على إصدار تقارير عن أرقام السطور التي تحتوي على أخطاء أثناء وقت التشغيل. ويتم عرض أرقام السطور بشكل افتراضي. والسبب الوحيد الذي يمكن أن تلغي فيها أرقام السطور هو فقط عندما تكون لديك عبارات طويلة وتريد رؤيتها على الشاشة دونما حاجة إلى تدوير الشاشة Scrolling إلى نهايتها.

خيار تبادل الشاشات Exhange Screens: يسبب هذا الخيار إزعاجاً لاداعي له عندما ترجع إلى برنامجك كل مرة ، وذلك بعرض شاشة برنامجك لوقت قصير جداً أمامك على الشاشة. ولعل شخصاً ما قد طلب إضافة هذا الخيار إلا أننا لا ننصح باستخدامه ولا نرى كبير فائدة له. وقد تم تشغيل هذا الخيار بشكل افتراضي ولذا ، فإننا ننصح بإيقاف تشغيله بسرعة كي لا يحتاج المستخدم إلى مراجعة طبيب العيون لما يسببه من إزعاجات للعيون.

بدّل الإدخال Swap on Input: لن يفعل هذا الخيار أي شيء مالم تأخذ بالنصيحة السابقة لإيقاف عمل خيار تبادل الشاشات السابق. فهو يبدل بين كل من شاشات برنامج كشف الأخطاء Debugger وشاشات البرنامج ذاته أثناء فترة انتظار البرنامج لعمليات إدخال البيانات (على سبيل المثال ، INKEY(0)). كما أن هذا الخيار هو في وضعية التشغيل بشكل افتراضي أيضاً هو الآخر.

خيار تتبع كتلة شيفرة code Block Trace: يعتبر هذا الخيار مفيداً لدراسة طريقة عمل كتل الشيفرة. وسيقفز برنامج كشف الأخطاء في كل مرة تقيّم فيها كتلة شيفرة ، سيقفز إلى السطر الذي تم إنشاء كتلة شيفرة عنده. وكما سيتبين في قسم كتلة الشيفرة فإن هذه الكتل يتم تقييمها دائماً من خلال نقطة تأسيسها وإنشائها ، بدلاً من تقييمها من المكان الذي أنت فيه في البرنامج. ونحن لا ننصح بإيقاف عمل هذا الخيار ، وإن فعلت ذلك فستتحمل أنت نتيجة عملك.

خيار شريط الأوامر **Menu Bar**: يعمل هذا الأمر على إخماد عرض شريط أوامر قائمة برنامج Debugger. وحتى في حالة توقيف عمل هذا الشريط ، يمكنك التعامل مع القوائم.

خيار أحادي اللون mono Display: يعمل هذا الخيار على التبديل بين اللون الأحادي والملون لعرض نوافذ برنامج Debugger.

الألوان Colors: يمكنك هذا الخيار من تفصيل ألوان برنامج كشف الأخطاء Debugger على ذوقك ورغبتك. كما يمكنك تغيير ألوان شيفرة المصدر و شيفرة PPO وإطارات النافذة و وخيارات القائمة. ولاحظ أن هذا الخيار لامعنى له إذا كنت قد اخترت اللون الأحادي في الخيار السابق ووضعته في وضعية التشغيل.

خيار عرض الحقل Tab Width: إذا كنت ممن يحبون استخدام مفتاح الجدولة الأمامية Tab بدلاً من الفراغات لإبعاد شيفرة مصدر برنامجك عن الهامش ، فيمكنك هذا الخيار من تحديد عدد الفراغات لاستبدالها بحقول الجدولة الأمامية. أما عدد المسافات المفترض في كل حقل فهو ٤ أربع مسافات فقط.

خيار مسار الملفات Path for Files: يمكنك هذا الخيار من تحديد مسار بحث بديل لملفات شيفرة المصدر source code.

خيار احفظ التجهيزات Save Settings: يمكنك هذا الخيار من حفظ تجهيزات برنامج كشف الأخطاء Debugger ، واستعادتها كما كانت لاحقاً. وتتضمن هذه التجهيزات كلاً مما يلي : كل الخيارات التالية ، حجم نوافذ برنامج Debugger ، وأية نقاط توقف تم تحديدها. ويمكن أن يوفر عليك هذا الخيار كمية كبيرة من الوقت. يمكنك حفظ تجهيزاتك هذه لبرنامج Debugger إلى ملف يسمى INIT.CLD والذي يتم تحميله فور تشغيل برنامج Debugger. انظر النقاط المبينة أدناه لمزيد من التفاصيل.

خيار أعد التجهيزات كما كانت Restore Settings: يعمل هذا الخيار على قراءة محتويات الملف الذي تم تجهيز ضوابط برنامج Debugger فيه مسبقا ، وهو ملف سكريبت script file.

قائمة اختيارات النافذة Window Menu

تمكنك هذه الخيارات من القيام بأى عمل تتخيله على النوافذ المختلفة لبرنامج Debugger. ومعظم هذه الخيارات غنية عن الشرح. ولابد طبعا من استخدام كل من مفتاح الجدولة الأمامية Tab ، ومفتاحي Tab-Shift للتنقل بين النوافذ بدلا من استخدام خيارات قائمة الاختيارات هذه. ويمكنك التركيز على أية نافذة بشكل مؤقت باستخدام مفتاح F2.

ولعل الخيارين الوحيدين اللذين يحتاجان إلى مزيد من الشرح و التفصيل هما: اصنع أيقونة Iconize ومربع Tile. فبينما يقوم اصنع أيقونة بتصغير النافذة الحالية بحيث تصبح على ارتفاع سطر واحد فقط بعرض عدة أعمدة ، ويمكن إعادتها إلى وضعها الأصلي بإنتقاء الخيار ذاته **Iconize** مرة ثانية. أما خيار **Tile** فيعيد النافذة على شكل المربعات المفترض. ولعل هذا يكون نافعاً بعد أن تغير أشكال النوافذ والشاشات بحيث لايمكن التعرف عليها بعد ذلك. فاختيار هذا الخيار يعيد النوافذ إلى وضعها السابق قبل التغيير.

وهناك عدد آخر من المفاتيح التي يمكنك استخدامها لتغيير حجم النوافذ وهي :

المفتاح	العمل
Alt-S	يصغر النافذة الحالية بمعدل سطر واحد
Alt-G	يكبر النافذة الحالية بمعدل سطر واحد
Alt -U	يحرك الإطار الموجود بين الأمر ونوافذ البرنامج إلى أعلى
Alt-D	يحرك الإطار الموجود بين الأمر ونوافذ البرنامج إلى أسفل

لاحظ أنك كلما حفظت التجهيزات في ملف script file لبرنامج كشف الأخطاء فسيتم حفظ الوضعية الحالية للنوافذ أيضاً. بل أن الاختبار السريع لهذا الملف سيكشف لك سلسلة كاملة من الأوامر المتعلقة بالنافذة ، وذلك لأنك في كل مرة تغير فيها تجهيزات النوافذ (تصغير ، تحريك ، تكبير ، إلخ..) سيحفظ البرنامج هذه التعديلات التي أجريتها في هذا الملف ، ولدى كتابة هذا الملف يتم حفظ كافة محتويات نافذة برنامج Debugger.

قائمة اختيارات المساعدة

تحتوي هذه القائمة على كل ماترغب الاستعلام عنه في برنامج كشف الأخطاء Debugger. ويجب أن تستخدمه بشكل مكثف للحصول على معلومات قيمة من هذا البرنامج الذي تتعامل معه.

استخدام نافذة الأوامر

يمكنك الكتابة في نافذة الأوامر مباشرة إذا أصبحت داخل البرنامج Debugger. فيمكنك إعادة أمر ما باستخدام مفتاح F3 كما يمكن استخدام سهمي الاتجاه إلى أعلى ↑ وإلى أسفل ↓ لمراجعة آخر الأوامر التي يتم إدخالها ، وهكذا.

وإن أشهر استخدام لنافذة الأوامر لعرض نتيجة تعبيرات كليبر (وهو: يتضمّن متغيرات استدعاء ووظائف، وثوابت). وتستخدم علامة الاستفهام "؟" لهذا الغرض. فعلى سبيل المثال : يمكنك كتابة "x ?" لمشاهدة محتويات المتغير X.

التفتيش Inspection

يلاحظ أن أحد أهم النواقص التي أخذت على برنامج كشف الأخطاء لكليبر إصدار Summer"87 هو عدم قدرته على تفتيش المصفوفات بسهولة. ولقد تم حل هذه المشكلة في كليبر 5.x باستخدام ميزة "تفتيش" وهي: ("??") . وإذا كتبت علامتي

استفهام متبوعة باسم المصفوفة المطلوب تفتيشها. وستفتح أمامك نافذة تفتيش في وسط الشاشة. اضغط على مفتاح Enter ، ثم استخدم مفاتيح أسهم الاتجاهات للتنقل مابين عناصر المصفوفة. فعلى سبيل المثال: إذا كانت لديك الوظيفة ()Directory مربوطاً ببرنامجك ، يمكنك كتابة الأمر التالي: "??directory()" لإنشاء مصفوفة لملف معلومات ومشاهدتها في نافذة تفتيش.

كما يمكنك استخدام أمر "تفتيش" على عناصر أخرى في المصفوفات ، ولكن أفضل استخدام له هو مع المصفوفات array والأهداف object. كما يجب أن تلاحظ أنه لايمكنك مشاهدة سوى عناصر المصفوفة أو المتغيرات الفورية instance variables ولن يمكنك تحريرها مباشرة من خلال نافذة الشاشة أو المشاهدة.

مختصرات سطر الأوامر

يمكن استخدام الكلمات التالية بدلاً من استخدام قوائم الاختيارات:

الكلمة المحجوزة	الوصف
animate	تشغيل برنامج ما في وضعية الرسم
bp	تجهيز نقطة توقف
callstack	عرض نافذة **Callstack**
delete	لمسح تجهيز واحد للنقطة ، أو بعض التجهيزات أو جميعها .
dos	انتقل إلى " دوس " دون الخروج من البرنامج الحالي
find	ابحث عن سلسلة حرفية في ملف حالي
go	يبدأ بتشغيل برنامج
goto	ينقل المؤشر إلى سطر محدد في البرنامج
help	يعرض شاشات المساعدة

الجدول مستمر من الصفحة السابقة....

الوصف	الكلمة المحجوزة
يقرأ أوامر برنامج كشف الأخطاء من ملف خطي (Script File)	inpnt
يدرج بعض تجهيزات نقطة ( ) أو جميعها	list
يبحث عن الحدوث التالي لسلسلة حرفية	next
يشغل/يوقف كتابة أرقام السطور في نافذة الشيفرة	num
يعرض شاشة برنامج	output
يبحث عن الحوادث السابق لسلسلة حرفية	prev
يخرج من برنامج اكتشاف الأخطاء وتصحيحها Debugger	quit
يعيد تجميل برنامج ما ، وتشغيله ، إلا أنه يبقي تجهيزات برنامج اكتشاف الأخطاء وتصحيحها على ما كانت عليه قبل الإيقاف	restart
يعود إلى برنامج اكتشاف الأخطاء وتصحيحها بعد مشاهدة ملف ما	resume
يضبط سرعة خطوة رسم	speed
ينفذ سطر برنامج حالي ويتوقف	step
يحدد نقطة متابعة	tp
يمكنك من مشاهدة ملف محدد	view
يمكنك من تجهيز نقطة مراقبة	wp

كما يمكنك ، إلى جانب استخدام الكلمات المفتاحية keyword المبينة أعلاه ، الوصول إلى أي خيار من خيارات برنامج كشف الأخطاء وتصحيحها Debugger بكتابة الحرف الأول من قائمة الاختيارات ، والحرف (أو الحروف) المتميزة الأولى من اسم ذلك الخيار المطلوب.

ونبين فيما يلي بعض الأمثلة عن هذه الطريقة:

f d	الخروج إلى غلاف "دوس" ، أو على الأقل يحاول ذلك !
m p	راقب المتغيرات العامة
m pr	راقب المتغيرات الخاصة
v s	شاهد التجهيزات العامة للنظام
o sa blah	احفظ الخيارات الحالية في ملف BLAH.CLD
o r blah	أعد الخيارات من ملف Blah.CLD كما كانت سابقاً (قبل التغيير)
r sp 50	أضبط سرعة الرسم على نصف ثانية
q	أخرج من برنامج اكتشاف الأخطاء وتصحيحها
v c	شاهد (أعرض) مجموعة الاستدعاءات (Callstack)
w i	ضع النافذة النشطة على شكل أيقونة
w t	أعد بناء المربعات في تجهيزات النافذة كما كانت عليه سابقاً بشكل مفترض

ملف الاستهلال INIT.CLD

لازلنا حتى الآن نستغرب ، ونعجب من عدم سماع كثير من مستخدمي كليبر عن هذا الملف علماً بأنه موضح تماماً وموثق في برنامج CA-Clipper Norton Guide ، كما أنه موضح تماماً أيضاً في شاشات مساعدة برنامج كشف الأخطاء Debugger. إذا حفظت خيارات برنامج Debugger في ملف بهذا الاسم ، فإن برنامجك سيحمل هذا الملف في كل مرة يتم تشغيله فيها بعد ذلك ، ويقوم بضبط تجهيزاتك طبقاً لها. وإن هذا العمل

يمكنك من توفير كمية لا بأس بها من الوقت وخاصة عندما تعلم تماماً أنك ستستخدم التجهيزات ذاتها للبرنامج الحالي. فعلى سبيل المثال ، يمكنك ضبط كل من حجم النوافذ ومواضعها ، والألوان ثم يمكن استخدامها لاحقاً كما هي في كل مرة تعيد فيها تشغيل البرنامج من جديد.

يبحث برنامج Debugger عن ملف INIT.CLD في الدليل الحالي ، فإذا لم يجده هناك فسيذهب محاولاً البحث عنه في الأدلة المحدد بأمر "المسار" Path.

محتويات ملفات الكتابة Script Files

قد لا يكون هذا الأمر بذاته ، إلا أنه يجب أن تحيط علماً بأنه يمكنك وضع مختلف الأوامر في ملفات الكتابة في برنامج Debugger. بل ، يمكنك في الحقيقة ، استخدام مختصرات الأوامر السابقة الذكر للتوصل إلى أية قائمة اختيارات من قوائم برنامج Debugger ، والتعامل معها مباشرة. لاحظ أنه لايمكنك ، مع ذلك الوصول إلى قائمة خيارات التشغيل Run فيما يتعلق بملف التشغيل الاستهلالي INIT.CLD (والاستثناء الوحيد لهذه القاعدة هو أنه يمكنك ضبط سرعة الرسم فقط).

وإنه كلما حفظت خياراتك ، ستتم كتابة تجهيز الألوان لبرنامج Debugger في الملف. وليست هذه الأمور واجبة على الإطلاق ، فيمكنك حذف عبارات الألوان من ملف الكتابة باستخدام محرر النصوص.

كما يجب أن تلاحظ أيضاً أنك إذا حفظت أية معلومات تتعلق بالنافذة ، كالحجم مثلاً في ملف الكتابة ، فإنه سيتم كتابة تاريخ أمر النافذة بأكمله في هذا الملف. ويمكنك رفع مستوى التنفيذ إلى الحد الأقصى بحذف الأوامر التي لاحاجة لك بها. فمثلاً : إذا تنقلت كثيراً بين النوافذ باستخدام مفتاح الجدولة الأمامية قبل حفظ خياراتك ، فسيكون لديك دون أدنى شك "النافذة التالية" (Window Next) أو "النافذة السابقة" (Window Previous) وعباراتها العديدة في ملف الكتابة.

المرجع السريع لمفاتيح وظائف برنامج Debugger

المفتاح	الوظيفة
F1	يعرض شاشات المساعدة
F2	يكبر/يصغر النافذة الحالية لبرنامج Debugger (هام)
F3	يكبر آخر أمر في نافذة الأوامر (هام !)
F4	يعرض شاشة برنامج (لاحاجة له عند استخدام أمر " قسم الشاشة ")
F5	يشغل البرنامج (عملي)
F6	يشغل شاشة منطقة العمل (عملي)
F7	يذهب إلى المؤشر (عملي)
F8	ينتقل خطوة خطوة داخل برنامج ما (لا تزد على ذلك)
F9	يجهز أو يحدد نقاط توقف على سطر المؤشر الحالي (عملي)
F10	(لا) تتابع هذا العمل (موفر للوقت)

إنشاء متغيرات باستخدام برنامج Debugger

كان كليبر 87 'Summer لاكتشاف الأخطاء Debugger يحتوي على خيارات في قوائم الاختيارات تمكنك من إنشاء متغيرات عامة أو خاصة بسرعة. ومع أنه لم يعد هناك مثل هذه الخيارات الواضحة التي تمكنك من القيام بمثل هذه الأعمال ، فلازال بإمكانك إنشاء

متغيرات خاصة من أي نوع من أنـواع البيانـات. ولابـد هنـا مـن اسـتخدام عـامل التعيـين المباشر في نافذة الأوامر. فمثلاً ، تبين العبارة التالية:

```
? X := 50
```

سيقوم عامل التعيين بإنشاء متغيراً خاصاً هو X ويجعله يبتدىء بالرقم ٥٠.

ويستحسن عدم استخدام هذه الميزة إلا لماما. بل لعل المرة الوحيـدة الـتي أعتقـد أنـه يمكـن استخدامها هي عند اكتشاف فقدان إسناد متغير ما في برنـامجك بحيـث يقضـي علـى هـذه المشـكلة. وبـدلاً من الخروج من برنـامج Debugger إلى برنـامج التحريـر ، أو إعـادة التجميع ، أو إعادة الربط ، فقد ترغب مؤقتاً بإنشاء "متغير مفقـود" علـى جنـاح السـرعة بحيث يمكنك على الأقل من تجاوز سطر محدد.

كما يمكنك أيضاً إنشاء مصفوفات ، وكتل شيفرة على جناح السرعة أيضاً داخل برنـامج Debugger. جرب مايلي:

١) شغّل برنامج Debugger باستخدام أي برنامج اختبار.

٢) اكتب مايلي في نافذة الأوامر:

```
? xxx :=  { 1, 2, 3  }
? yyy :=  { | | xxx[1]  }
ALT-m  v
```

وستشاهد في نافذة المراقبة المصفوفـة الـتي قمـت بانشـائها XXX وكذلـك كتلـة الشـيفرة YYY.

٣) تحول إلى نافذة المراقبـة باسـتخدام المفتـاح Tab. علّـم XXX واضغـط علـى مفتـاح Enter ثلاث مرات ، وسـتصبح علـى العنصـر الأول مـن المصفوفـة XXX. إكتـب "{ 1,2,3 }" ، وبعد ذلك اضغط على مفتاح Enter وسترى أن ذاك العنصر سيتحول فوراً إلى مصفوفة يمكنك الأنتقال بين عناصرها وتفتيشها كما يحلو لك بعد ذلك.

فإذا كان لديك الوظيفتان ( )DIRECTORY أو ( )DBSTRUCT مربوطتين ببرنامجك فيمكنك عندئذ إنشاء مصفوفات سريعة on-the-fly بكتابة تلك الوظائف بدلاً من كتابة المصفوفات ذاتها.

ويعمل هذا الأمر بهذا الشكل لأنك كلما غيّرت قيمة متغير داخل برنامج كشف الأخطاء Debugger فإن إدخال بياناتك هذه سيمر من خلال مجمع ماكرو كليبر. ولذلك ، فتجد أن هذه العملية هي واقعية وحقيقية.

الربط باستخدام البرنامج Debugger

إذا أردت القيام بإجراء عملية الربط باستخدام البرنامج Debugger ، فإن هذا العمل سيتم بشكل مماثل تماماً لما هو عليه في كليبر Summer'87. ابحث عن الملف CLD.LIB ولكن حاول ألا تنخدع بربطه على أنه "مكتبة" بل يجب ربطه على أنه ملف "هدف" ويبين الأمر التالي هذا الربط:

```
rtlink fi myprog , cld.lib
```

وعند تضمين برنامج Debugger في برنامجك فيمكنك تنشيطه بالضغط على مفتاحي D- Alt ، كما هو الحال أيضاً في كليبر Summer'87 ، إلا أنه على خلاف ماهو عليه الحال في Summer'87 حيث يمكنك إيقاف عمل هذين المفتاحين D- Alt في كليبر 5.x ، للوظيفة ()ALTD. فإذا مررت صفراً للوظيفة ()ALTD فستوقف بهذا عمل البرنامج Debugger. وإذا أردت تشغيله ثانية ، مرر واحداً (1) للوظيفة ()ALTD. فإذا قمت باستدعاء الوظيفة السابقة دون ذكر متغيرات فإنه سيستدعي برنامج Debugger مفترضاً أنه لازال نشطاً وفي وضعية التشغيل.

ويبين البرنامج التالي كيف تحد من استعمال برنامج Debugger للمستخدمين الذين هم من مستوى الأمن ١٠٠ فما فوق.

```
function main ( sec_level )
```

```
if  sec_level  == NIL  .or.  val(sec_level)  <  100
   altd (0)
endif
```

الخيار ()DISBEGIN و ()DISPEND داخل برنامج Debugger

إذا استخدمت أحد هذين الخيارين في شيفرة المصدر الخاصة بك ، يجب أن تنتبه إلى أن برنامج Debugger سيتجاهلهما تماماً. وسيتم توجيه كافة مخرجات الشاشة إلى الشاشة العادية. وإن هذا الأمر ضروري لأن أجزاء من برنامج اكتشاف الأخطاء وتصحيحها قد كتبت باستخدام كليبر ذاته ، وبهذا فإن هذين الخيارين سيؤثران كذلك على الإخراج (ويسببان ألماً وإزعاجاً لا نهاية لهما).



المعالج الأولي Preprocessor

يحتوي كليبر 5.x على معالج أولي ضمني ، والذي يمكن أن تفرد له مجلدات للحديث عنه على أن كليبر 5.x هو لغة تطوير معترف بها. وسنبين لك في هذا القسم كيفية استخدام "المعالج الأولي" بحيث تتمكن من تشغيل برامجك بسرعة عالية ، وبحيث تصبح البرامج أسهل قراءة وذات نهاية مفتوحة.

يقوم المعالج الأولي بإعداد عدة ترجمات لبرنامجك قبل تجميعه الحقيقي. إلا أنه يفوق إلى حد كبير آلية "البحث والاستبدال". وتتضمن عملياته كلاً مما يلي:

- الترجمة (البسيطة والمعقدة)
- تضمين ملفات أخرى (تسمى في كليبر "ملفات الترويسة" ، ويشار إليها عادة بالنهاية CH.)
- تجميع شرطي لكتل معيَّنة من الشيفرة (والذي يعتبر أمراً رائعاً فيما يتعلق باكتشاف الأخطاء البرمجية وتصحيحها/أو للنسخ الخاصة بالعرض فقط).

وكما أشرنا آنفاً ، بما أن المعالج الأولي هو ضمني داخل مجمِّع كليبر ، فإنك ستقوم باستخدامه تلقائياً ، عرفت ذلك أم لم تعرفه ، لذلك يستحسن أن تتعلم كيفية استخدامه بحيث تتمكن من الاستفادة منه إلى الحد الأقصى.

الثوابت الظاهرة Manifest Constants

هذه الثوابت هي معرِّفات يعمل بموجبها المعالج الأولي. ولهذه الثوابت فوائد جمّة بما في ذلك تحسين درجة القراءة ، وتحسين سرعة التنفيذ ، والتجميع الشرطي.

تحسين درجة القراءة

يمكنك استبدال اسماء ذات معنى مكان الأرقام التي لا معنى لها باستخدام المعالج الأولي. وتسمح لك عبارة **define#** بإعلان الثوابت الظاهرة. وأما العبارة المستخدمه في ذلك فهي كما يلي:

```
#define <identifier> [ <value> ]
```

فإذا سبق أن حددت القيمة **<value>** فإن المعالج الأولي سيحدد مواقع كافة المعرّفات **<identifier>** الموجودة في شيفرة المصدر الخاصة بك ، ويستبدلها بالقيمة (بحيث يقارن عملية البحث والاستبدال).

ويجب الانتباه إلى أن تبديل المعالج الأولي للثوابت الظاهرة هو طبقاً للحالة ويتحسسها وننصح أن تتجنب المشاكل بالتزام التسمية المصطلح عليها في لغة **C** أي: استخدام الحروف الكبيرة (الأنجليزية) لكافة الثوابت االظاهرة كما تجدر الملاحظ إلى أن **<value>** هي قيمة اختيارية ، وسنبين لاحقاً كيف، ولماذا نحدد الثوابت الظاهرة دون استعمال هذا المتغير.

كما أنك لست بحاجة للإطلاع على مخرجات المعالج الأولي (بصرف النظر عن أنك لاتستطيع أن تفعل بها أي شيء). إلا أننا نقترح إذا كنت مبتدئاً باستخدام كليبر ، أن تستخدم هذا الخيار **P/** بشكل كبير في الأشهر الأولى أثناء التعامل مع المعالج الأولي. وسينشيء هذا الخيار ملفاً خاصاً بالمعالج الأولي ، وسيحمل اسم ملفك **PRG.** ذاته ، إلا أن نهايته ستكون **PPO.** ، وبمراجعة هذه الملفات بدقة سيمكنك أن ترى أن المعالج الأولي يفعل ما يجب عليه القيام به لشيفرة المصدر الخاصة بك ، وبهذا يمكنك تعلم المزيد عن الأعمال الداخلية التي يقوم بها كليبر.

وإليك المثال الأول من شيفرة المصدر:

البرنامج الأصلي **(PRG.)**

```
# define  K_DOWN        24
# define  K_UP           5
# define  K_LEFT        19
# define  K_RIGHT        4

IF  keypress  == K_DOWN  .OR.   keypress == K_UP   .OR.  ;
    keypress  == K_LEFT   .OR.  keypress ==  K_RIGHT
```

مخرجات المعالج الأولي (PPO.)

```
IF  keypress == 24  .OR.   keypress ==  5  .OR. ;
    keypress == 19  .OR.  keypress == 4
```

ملاحظة

يقوم المعالج الأولي بحذف كل ما لم يترجم ، ويترك سطوراً فارغة مكانها بحيث ترى عدة أسطر فارغة في بداية ملف PPO. .

وكما ترى ، فإن عملية البرمجة تصبح أكثر جلاء ووضوحاً عندما تستبدل الأرقام بالكلمات بحيث ترى مايحدث تماماً. وقد تكون ذا قدرة خيالية على استذكار الأرقام وحفظها وإعادتها بأكملها كما هي في قائمة القيم ( )INKEY ، عن ظهر قلب. إلا أن استخدام الكلمات بدلاً من الأرقام سيساعد من يخلفك للقيام بأعمال الصيانة على البرامج التي أعددتها.

كما أنك باستخدام المعالج الأولي ستوفر كثيراً من الوقت لعدم الحاجة إلى إضاعته بحثاً عن قيم ()INKEY. بل إن كليبر يشتمل على ملف ترويسة يُسمى ()INKEY يحتوي على "بيان ثوابت" لكافة القيم المطابقة لـ: ( )INKEY والتي يصعب تذكرها. وبما أن هذه "الثوابت " تستخدم مصطلحات تسمية ثابتة ومتعارف عليها (مثل **,K_ENTER** **K_TAB, K_Ctrl**) فمن المحتمل جداً ألا تحتاج لمراجعة هذا الملف أيضاً. وماعليك إلا أن تضمّن هذا الملف ( )INKEY في برنامجك باستخدام أمر التضمين #include، وتضمين الملف INKEY.CH ، واستخدم الخيار *_K لثوابت البيان كلما احتجت لذلك. وسيقوم المعالج الأولي بترجمة الكلمات إلى أرقام بدلاً عنك.

وهناك سبب وجيه آخر لاستخدام "ثوابت البيان" وهو أن المعالج الأولي سيسمح لك باستخدام ٣٢ رمزاً (حرفاً) كثابت بيان (بدلاً من الحد الاقصى وهو ١٠ رموز لاسم المتغير) ، ويمكنك إيضاح التسمية أكثر باستخدام ٣٢ حرفاً بدلاً من ١٠ حروف فقط أليس كذلك.

المصفوفات مقابل متغيرات الذاكرة

إذا استخدمت أياً من متغيرات الذاكرة من نوعي Private أو Public لحجز قيم الحقل لتعديلها ، فيمكنك توفير الذاكرة الثمينة باستخدام المصفوفات Arrays بدلاً من تلك المتغيرات المكلفة. ويعود هذا إلى أنك تنقص عدد الرموز التي يحتوي عليها برنامجك. وبهذا تنقص حجم جدول الرموز في البرنامج.

إلا أن هناك محذوراً واحداً فقط لاستخدام المصفوفات بدلاً من المتغيرات وهو أنه غالباً ماتؤدي إلى إضعاف القدرة على قراءة شيفرة المصدر ، إلا أننا باستخدام المعالج الأولي فإننا نحصل على أفضل فوائد الطريقتين معاً: فيمكن استخدام المصفوفة ، ولكن يمكن أيضاً إنشاء " ثوابت ظاهرة " لتعطي كل عنصر من عناصر المصفوفات معنى أكثر وضوحاً وجلاءً.

```
local  aMemvars[ 8 ]
# define  MFNAME              aMemvars[1]
# define  MLNAME              aMemvars[2]
# define  MADDRESS            aMemvars[3]
# define  MCITY          aMemvars[4]
# define  MSTATE              aMemvars[5]
# define  MZIP                aMemvars[6]
# define  MFRIEND             aMemvars[7]
# define  MBIRTHDATE          aMemvars[8]
```

وبعد ذلك يمكنك كتابة GET's حيث ستكون قادراً على معرفة ماذا يحدث في برنامجك.

```
@   7 ,  28   get   MFNAME
@   8 ,  28   get   MLNAME
@   9 ,  28   get   MADDRESS
```

```
@ 10 ,  28  get  MCITY
@ 11 ,  28  get  MSTATE
@ 12 ,  28  get  MZIP
@ 13 ,  28  get  MFRIEND picture  "Y"
@ 14 ,  28  get  MBIRTHDATE
```

بينما تكون مطابقات المصفوفة رموزاً بالغة التعقيد ولايفهم منها شيء:

```
@  7 ,  28  get  aMemvars[1]
@  8 ,  28  get  aMemvars[2]
@  9 ,  28  get  aMemvars[3]
@ 10 ,  28  get  aMemvars[4]
@ 11 ,  28  get  aMemvars[5]
@ 12 ,  28  get  aMemvars[6]
@ 11 ,  28  get  aMemvars[7]  picture  "Y"
@ 14 ,  28  get  aMemvars[8]
```

وعندما تصبح أكثر ارتياحاً عند استخدام المصفوفات المتداخلة ، ستصبح " ثوابت البيان " لاغنى لك عنها ، إذ أن استخدام هذه الثوابت لتعريف تركيبة مصفوفاتك المتداخلة منذ البداية ، سيحميك من الضياع في متاهات الإشارات المرجعية لعنصر المصفوفة غير الثابت.

تحسين سرعة التنفيذ

إذا عدنا إلى المثال الأول في بداية هذا النقاش (اختبار ضغط المفاتيح) ، فيمكن أن نعالج هذه المشكلة بطريقة أخرى وهي: تعريف " متغيرات: بدلاً من " ثوابت بيان " ، على النحو التالي:

```
K_DOWN   =   24
K_UP     =   5
K_LEFT   =   19
K_RIGHT  =   4
```

ولقد كانت هذه الطريقة هي الوحيدة في الإصدارات السابقة من كليبر. وقد استخدم كثير من المطورين هذه الطريقة لتحسين درجة قراءة برامجهم (والتي يمكنك أن يطلق عليها

ثوابت البيان الزائفة pseudo-manifest constants). إلا أن هناك محذورين لهذه الطريقة أيضاً بالمقارنة مع المعالجة الحقيقية لثوابت البيان ، وهما:

١- يتم الإحتفاظ "بالثوابت الزائفة" في "جدول الرموز" بدلاً من حلّها أثناء التجميع. وهذا يعني أنه كلما أُشير إليها أثناء تنفيذ البرنامج ، يجب البحث عن قيمها في "جدول الرموز" المناسب ، ومع أن البحث لا يستغرق وقتاً طويلاً ، إلا أنه يبطىء عمل البرنامج عدة دورات ، ولنقم معاً باختبار بسيط على النحو التالي:

```
* using pseudo - constants
TEST = 5
for xx = 1 to 1000
   Y = TEST
next

* using manifest constants
#define TEST 5
for xx = 1 to 1000
   Y = TEST
next
```

ونلاحظ أن الحلقة الثانية يتم تنفيذها بسرعة تزيد ١٠٪ عن الحلقة الأولى. والميزة الثانية هي أن حجم برنامجك سيصبح أصغر لأنه لم تعد هناك حاجة لإدخال جدول رموز لثوابت البيان التي تستخدمها (كما هو الحال في استخدام متغيرات الذاكرة التقليدية).

٢- إن رموز الثوابت هي عرضة لتغيير العرضي خلال إعداد البرنامج. فمثلاً: ما الذي يمنعك من التنقل ما بين النوع الرقمي والنوع الحرفي ؟ لاحظ المثال التالي:

```
* at the top of the program
K_RIGHT := 4

* 3000 lines further down
K_RIGRT := CHR(4)
```

ما الذي سيحدث في المرة التالية عندما يشير برنامجك إلى K_RIGHT ؟. انظر ماذا سيحدث ! وقد تعترض هنا قائلاً: "لايحتمل أن أقع في مثل هذا الخطأ !". وإني أقرك على

هذا ، فأنت مبرمج جيد دون شك. إلا أنه يحتمل أن يقوم بعض المبرمجين الآخرين بإجراء تعديلات على البرنامج الذي أعددته أنت ، وقد لايكون هؤلاء مثلك في الدقة والجودة.

وطالما لازلنا نتكلم عن موضوع سرعة التنفيذ ، فيمكنك أيضاً أن توفر المزيد من الوقت باستبدال استدعاءات الوظيفة السطرية (ذات السطر الواحد) بماكرو المعالج الأولي. وهذا النوع من الماكرو ليس مماثلاً ، أو حتى قريباً من الماكرو التقليدي لقاعدة البيانات dBASE. ولذلك ، يمكن تمييز ماكرو المعالج الأولي عن ماكرو قاعدة البيانات بتسميتها "وظيفة برمجية زائفة pseudo-functions " بحيث نعطيه شيئاً من الوضوح والتحديد. وتشبه عبارة "وظائف البرمجة الزائفة" عبارة " ثوابت البيان " إلى حد كبير ، ويمكن كتابتها على النحو التالي:

```
#define  <function> (  [  <argument list>  ]  ) <expression>
```

وسيتتبع المعالج الأولي كل "وظيفة" <Finction> من الوظائف التي يحتوي عليها برنامجك، ويستبدلها بالتعبير <expression>. وإذا حددت (قائمة المتغيرات) < argument list>، فسيتم استبدال هذه بالتعبير <expression> بناء على الاسماء التي تعطيها لها في (قائمة المتغيرات) <argument list> فعلى سبيل المثال:

```
#define  whatever(exp1,   exp2 ) exp1  +  exp2
x   :=  whatever( "ABC" , "123" )
```

سيتم معالجتها لتصبح على الشكل التالي:

```
x := "ABC"  +  "123"
```

وبما أننا حددنا القائمة (exp2 , exp1) ، فإن EXP1 سيأخذ قيمة "ABC" والتعبير EXP2 سيأخذ القيمة "ABC" بحيث يقوم المعالج الأولي باستبدالهما بالتعبير exp1 + exp2.

ولابـد مـن اتبـاع بعـض القواعـد البسـيطة إذا أردت تحديـد قائمـة المتغـيرات <argument list> ، وهي:

- يجب عدم وضع مسافات فارغة بين اسم الواجب والقوس المفتوح:

```
#define  whatever( exp1 ,   exp2 )   exp1 + exp2       // fine
#define  whatever  ( exp1 ,   exp2 )   exp1 + exp2       // nope
```

- يجب اتباع قائمة المتغيرات بقوس لإغلاقها.

ولنحاول الآن كتابة "وظيفة برمجية زائفة" MAXY(). والتي تقبـل ثلاثـة متغـيرات رقميـة ويعيد أعلاها قيمة. ونبدأ أولاً بكتابتها بشكل عادي UDF:

```
x 1   : =  500
x 2   : =  1000
for  x x  : =  -1000  to  1000
      yy   : =  MaxY  ( x1,  xx,  x2 )
next

function MaxY( a , b , c )
return max(max( a , b ), c )
```

والآن لنحاول كتابتها من جديد على شكل"وظيفة برمجية زائفة pseudo-function":

والبرنامج الأصلي هو : (.PRG)

```
#define  MAXY  (a , b , c )   MAX(MAX( a , b ) ,  c )
x 1   : =  500
x 2   : =  1000
for  x x  : =  -1000  to  1000
   yy  := MAXY ( x1 , xx , x2 )   // note upper - case
next
```

وأما ملف مخرجات المعالج الأولي فهو (.PPO):

```
x1  := 500
x2  := 1000
for  x x  :=  -1000 to 1000
   yy  := MAX(MAX( x1 , xx ) , x2 )
next
```

ويتم تنفيذ "الوظيفة البرمجية الزائفة" بسرعة تزيد ٢٥٪ عن استدعاء الوظيفة (حتى عند استخدام متغيرات محلية Local في هذه الوظيفة ، وذلك عند استخدام الإعلان الخاص PRIVATE ، وستكون السرعة النسبية لتلك الوظيفة أبطأ من هذه بكثير). والسبب الرئيسي لتحسين سرعة التنفيذ هو أن استدعاء الوظيفة يضيف شيئاً ، ولو قليلاً من الوقت الإجمالي للتنفيذ. ويجب أن يحفظ كليبر مكانك الحالي في تلك القائمة الإجمالية التراكمية ، ثم يقفز إلى مكان وجود تلك الوظيفة في الذاكرة. ثم يعيد كليبر ذلك المكان إلى وضعه الطبيعي عند الإنتهاء من تنفيذ تلك الوظيفة. وإن ترتيب كل هذه الأمور يزيل الاختناقات غير الضرورية على مستوى لغة الآلة في البرمجة.

كما تعطينا "وظائف البرمجة الزائفة" أيضاً المزيد من السعة بحيث نتمكن من القيام بما يجب علينا القيام به من أعمال مختلفة. ولابد من الانتباه التام أثناء توسيع قائمة المتغيرات <argument list> بحيث نقوم بالجمع القوسي بشكل دقيق ومناسب. خذ بعين الاعتبار خيار الضرب ( )Times ، وهو وظيفة برمجية بسيطة تقبل قسمتين رقميتين وتضربهما ببعضهما. مثال:

البرنامج الأصلي (PRG.):

```
#define  TIMES( a,  b ) a  *  b
w   := 5
x   := 4
y   := 3
z   := 2
t    := TIMES( w  +  x,  y + z )
```

أما ملف المعالج الأولي ، فهو (PPO.):

```
t  := w + x * y  +  z
```

وطبقاً لقواعد أولوية العوامل الرياضية ، فإن الضرب سيحدث قبل الجمع. ويحتمل أننا لانريد هذه النتيجة. فبدلاً من الحصول على نتيجة ٤٥ ، من جراء إجراء العملية التالية ((3+2) * (5+4)) ، فإن المتغير T ستعين له القيمة 19 (2+ (4*3)+5) إذ أنك أهملت الأقواس ، وقام المعالج الأولي باتباع الذي طلبته منه ، والآن ، لنحاول تصحيح هذه العملية:

```
#define TIMES( a, b ) (a) * ( b)
```

نسخة العرض (Demo) وبرنامج Debugger

لاشك أننا استخدمنا تعليمة (أمر) اكتشاف الأخطاء وتصحيحها Debug في برنامجنا بين آونة وأخرى. مثال:

```
debug  := .T.
*
* elsewhere in the program
if  debug
    ? "procname( )    =   " ,  procname( )
    ? "procline( )    =   " ,  procline( )
    ? "readvar( )     =   " ,  readvar( )
    ? "memory(0)      =   " ,  memory(0)
    ? " x =  " , x
    ? " y =  " , y
    ? " z =  " , z
endif
```

وبالطريقة ذاتها ، يمكن بشكل عام وضع شيفرة داخل برنامج ما بحيث يمكنك توزيع نسخة استعراضية للبرنامج على عدة عملاء يحتمل أن يكونوا مهتمين بهذا البرنامج ، وذلك على النحو التالي:

```
if  demo
    ? " This demo will only access 50 records "
    max_rec  : =  50
else
    mex_rec  : =  5000000000
endif
```

ومع أنه لن يتم تنفيذ كتل الشيفرة هذه إلا بشكل مشـروط ، فإنـه سيتم تجميع البرنـامج بشكل غير مشروط. ستتجمع كلها في الوحدات البرمجيـة الهدفيـة ، وبالتـالي ، ستكون في الملف التنفيذي EXE. ، ولا شك أن في هذا ضياعاً كبيراً للذاكرة.

ولحسن الحظ ، فإن المعالج الأولي preprocessor يعطينا القدرة على تجميع برامجنا بشـكل مشروط. فقد أشرنا سابقاً إلى أنه يمكن تحديد ثوابت بيان دون خيـار القيمـة <Value> ، وهنا هو المكان الذي يمكن استخدام هذا الخيار بـالضبط. وعنـد وجـود هـذا الخيـار سيتم توجيه المعالج الأولي لتجميع (أو عدم تجميع) أقسام محددة مـن البرنـامج الأصلـي ، وذلـك على النحو التالي:

```
#define <identifier>
```

ولايحتاج خيار <identifier> إلى قيمة ، بل كل ما في الأمر ، أن نضعه هناك كما هو. إلا أن هذا لن يكون مفيداً مالم تسـتفد مـن هذيـن الخيـارين الإرشـاديين ifdef# و ifndef# ، حيث يقوم الخيار الأول ifdef# بإرشاد المعالج الأولي إلى أنه عند وجود "محدد identifier " معيّن يجب أن يقوم بتجميع كتلة البرنامج التالية، وأما إذا وجد الخيار ifndef# لايجمّعه (عكس الخيار السابق)، بل يوجه المعالج الأولي لتجميـع الكتلـة التاليـة مـن شـيفرة المصـدر فقط إذا لم يوجد المحدد.

والآن ، لنحاول شرح المثال الـذي يتنـاول الأمـر DEBUG ثانيـة باسـتخدام الإرشـادات التالية:

البرنامج الأصلي (PRG.):

```
#define  DEBUG
#ifdef  DEBUG
    ? "procname( )  =  " ,  procname( )
    ? "procline( )      =  " ,  procline( )
    ? "readvar( )       =  " ,  readvar( )
    ? "memory(0)      =  " ,  memory(0)
    ? "x =  " , x
    ? "y =  " , y
    ? "z =  " , z
```

```
#endif
```

ملف المعالج الأولي (.PPO):

```
Qout("procname( ) =    ", procname( ) )
Qout("procline( )    =    ", procline( )   )
Qout("readvar( )     =    ", readvar( ) )
Qout("memory(0)   =    ", memory(0)  )
Qout(" x =  " , x  )
Qout(" y =  " , y  )
Qout(" z =  " , z  )
```

أما إذا لم يتم تعريف ثابت بيان DEBUG في defined# ، فستكون مخرجات المعالج الأولي على النحو التالي:

(whitespace) (مساحة فارغة)

والآن يمكنك أن تترك كافة تعريفات DEBUG داخل برنامجك دون أن تخشى جعل الملف التنفيذي للبرنامج كبيراً دون داعٍ وكل مايجب عليك أن تفعله هو أن تحدد define# DEBUG عندما تريد استخدامه ثانية.

ولعلك تفكرٌ هنا أيضاً وتتساءل: عندما يكون لدينا عبارة شرطية IF ، وعبارة إنهائها Endif. لابد أن يحتمل أيضاً وجود عبارة ELSE. ولاشك أنك محق في ذلك ، إننا نستخدم مثل هذه العبارة لتنظيف مثال نسخة الاستعراض على النحو التالي:

البرنامج الأصلي (.PRG):

```
#define DEMO

#ifdef  DEMO
    ?  " This demo will only access 50 records "
    max_rec  :=  50
#else
    max_rec  :=  5000000000    //  mammoth file
#endif
```

ملف مخرجات المعالج الأولي (.PPO):

```
Qout (   " This demo will only access 50 records "  )
 max_rec  :=  50
```

أما إذا أردنا حذف تعريف العرض DEMO ، فستكون مخرجات المعالج الأولي على النحو التالي:

```
max_rec  :=  5000000000
```

وبطريقة مماثلة ، فإن الخيار ifndef# يسمح بالتجميع الشرطي بناء على عدم وجود ثابت بيان ، وذلك على النحو التالي:

البرنامج الأصلي (PRG.):

```
#ifndef  REALTHING
   ?    " This demo will only access 50 records "
    max_rec  :=  50
#else
   max_rec  :=  5000000000
#endif
```

ملف مخرجات المعالج الأولي (PPO.):

```
Qout(   " This demo will only access 50 records "  )
 max_rec  :=  50
```

وهناك إرشادٌ آخر في هذه المجموعة يحتمل أن يكون ذا فائدة كبيرة لك ، وهو الخيار undef# وهو يحذف (يلغي التحديد) محدِّدا ما identifier.

ولهذا عدة أغراض ، أولها كونها حدُّ التجميع الشرطي لقسم من برنامجك الذي تعده على النحو التالي:

البرنامج الأصلي (PRG.):

```
#define  DEMO

#ifdef  DEMO
   max_rec  := 50
   max_calls  :=  100

#else
    max_rec  := 50000000
   max_calls  :=  100000
```

```
#endif
#undef  DEMO

#ifdef  DEMO
   max_times  := 25
#else
   max_times  := 200
#endif
```

أما ملف مخرجات المعالج الأولي (PPO.) (لقد تم تجاهل معظم السطور الفارغة):

```
max_rec  := 50
max_calls     :=  100

max_times     := 25
```

لاحظ ماذا حدث في نهاية كتلة #ifdef..#else..#endif وذلك لأنك ألغيت تحديد خيار محدد DEMO. ولذلك ، فإن المعالج الأولي جمعها شرطياً وكأنك كنت تستخدم برنامج demo.

والمثال الآخر حين تريد إعادة تحديد " ثابت بيان " ، سينتج هذا تحذير تجميع ما لم تلغ تحديده أولاً ، كما هو في المثال التالي:

```
#define  DEMO

#ifdef  DEMO
   max_recs  := 50

#else
    max_recs  := 10000
#endif

#undef  DEMO         // remove to make compiler whine !

#define  DEMO  .T.
```

عند استخدام الخيار #define " ثابت بيان " سيكون مرئياً من ذاك السطر إما إلى نهاية ذلك البرنامج ، أو حتى يتم إلغاء التحديد باستخدام أمر خيار #undefine. وتنطبق هذه

القاعدة أيضاً على "الثوابت الظاهرة" في ملفات الترويسة التي تضمنها باستخدام خيار #include (والذي سنبينه قريباً). وإن الاستثناء الوحيد لهذه القاعدة هو "الثابت الظاهر" المحددة باستخدام خيار #define في ملف الترويسة STD.CH (أو ملفات قواعد قياسية بديلة تم تحديدها باستخدام خيار u/ للتجميع).

خيار التجميع D/

يمكنك تحديد "الثوابت الظاهرة" باستخدام خيار #define أثناء وقت التجميع باستخدام خيار التجميع الذكي جداً ، واسمه D/. إذ يمكنك هذا من تغيير " ثوابت البيان " الموجودة في برنامجك دون أي حاجة لتغيير شيفرة المصدر ذاتها. ويمكنك إما أن تنشيء "ثوابت ظاهرة جديدة " ، أو إذا شئت استخدام توجيه أو إرشاد معين لخيار #ifndef تجاوز الثوابت الموجودة في البرنامج.

والآن ، لنراجع معاً عبارة استخدام خيار التجميع d/ :

```
clipper progname /d<ID> [ = <VAL> ]
```

حيث يمثل <ID> اسم " ثابت البيان " ، ويمكنك تعيين قيمة <VAL> بشكل اختياري لثابت البيان بأن تتبع <ID> بإشارة = ثم القيمة المطلوبة. فمثلاً: في الجزء الأخير من البرنامج السابق ، كان بإمكاننا حذف عبارة #define DEMO وتجميع البرنامج على النحو التالي:

```
clipper   test /dDEMO
```

وسيعطي هذا الأمر الأثر ذاته ، ولكن بالفائدة اللطيفة الإضافية وهي أن المبرمج لم يحتج أن يلمس برنامجه إطلاقاً بأي تغيير.

ويمكنك أيضاً ، وبشكل اختياري تعيين قيمة للمحدِّد. ولنفترض أنك تريد إنشاء مصفوفة وبدايتها بحجم معيّن ، فإن هناك أموراً أخرى مثل حلقات FOR...NEXT تعتمد هي

الأخرى على حجم المصفوفة ، وتريد تغيير كافة هذه الإرشادات المرجعية بوقت واحد ، فإن أسهل طريقة لتحقيق هذه العملية هي تحديد محدِّدٍ (أو "ثابت بيان") في أول برنامجك ، على النحو التالي:

```
#define ELEMENTS 500
local a[ELEMENTS], total , x
for x := 1 to ELEMENTS
    total += ( a[x] := x )
next
```

والآن ، لنفترض أنك تريد تغيير عدد العناصر دون تغيير البرنامج ذاته ، فيمكنك أن تفعل ذلك بسهولة باستخدام مفتاح d/. وإن تجميع شيفرة المصدر الخاصة بك باستخدام سطر الأوامر التالي ، سينتج عنه مصفوفة (وعدّاد حلقات) بألف بدلاً من ٥٠٠.

```
clipper myprog /dELEMENTS=1000
```

فإذا جمّعت هذا المثال فإنك ستحصل على رسالة خطأ تجميع "إعادة تحديد لخيار التحديد" (redefinition of #define). ولاشك أن هذا أمر معقول إذ أنك تقوم بتحديد العناصر ELEMENTS مرتين ، الأولى: عند تشغيل المجمّع ، والثانية: في البرنامج ذاته في المكان الذي حددته فيه بشكل أصلي ، فيجب الانتباه لذلك وعدم الوقوع في مثل هذا الخطأ:

وإليك فيما يلي حل بسيط لهذه المشكلة:

```
#ifndef ELEMENTS
   #define ELEMENTS 500
#endif
local a[ELEMENTS] , total , x
for x := 1 to ELEMENTS
   total += ( a [x] := x )
next
```

وسيخبر هذا الأمر المعالج الأولي أن يحدد العناصر ELEMENTS فقط إذا لم يسبق تحديدها.

وكلما استخدمت "الثوابت الظاهرة" في برنامجك ، إستطعت توفير مزيد من الوقت باستخدام خيار التجميع D/.

ملفات الترويسة Header Files

والآن ، وبعد أن أصبحت جاهزاً لبناء مجموعة رائعة من ثوابت البيان الخاصة بك ، فلا بد أن تعرف كيف تفرقها عن برنامجك ذاته. إن ملفات الترويسة ، (والتي تعرف أيضاً باسم "INCLUDE File" ملفات التضمين) هي أفضل مكان تحفظ فيه "ثوابت البيان" manifest constants والأوامر المعرفة من قبل المستخدم user-defined commands. وتحمل ملفات الترويسة عادة في كليبر النهاية ".CH". مثلاً ، بدلاً من أن تضع هذه جميعها في بداية ملف كل برنامج يستخدمها ، يمكن أن تقوم بما يلي:

```
#define  CRLF       chr(13)+chr(10)
#define  MAXY( a, b, c )  MAX(MAX( a, b ) , c )
#define  NETERR_MSG    "Network error, could not add/edit at this time"
```

فيمكنك أن تضعها جميعاً في ملف ترويسة CH. ، ثم تقوم بتضمينها في برنامجك بكل بساطة باستخدام الخيار include# على النحو التالي:

```
#include   "mystuff.ch"
```

ويقوم التوجيه include# ، وهو واضح بذاته ولايحتاج شرحاً ، بتضمين محتويات ملف الترويسة أثناء وقت التجميع. ويجب أن تحيط اسم ملف الترويسة بعلامتي تنصيص قبله وبعده دائماً ، كما يجب تحديد النهاية أيضاً. ولكن أيضاً أن تحدد السواقة والمسار ، ولكنك إذا لم تفعل ذلك فإن المعالج الأولي سيبحث عنهما وفق الترتيب التالي:

- الدليل الحالي
- الأدلة المحددة باستخدام خيار المجمّع i/

■ الأدلة التي إدرجت في بيئة متغيرات INCLUDE والتجهيز المقترح لهذه القائمة (SET INCLUDE=C:\CLIPPER5\INCLUDE)

مع أن ملفات الترويسة تحتوي عادة على ثوابت بيان وأوامر مسبقة التحديد من قبل المستخدم ، وقد تحتوي أيضاً على برامج عادية أخرى (ماعدا ملف الترويسة STD.CH وأي ملف قواعد قياسية بديل آخر).

ومع هذا ، فإننا لا نشجع القيام بمثل هذا العمل لأنه يجعل مستوى اكتشاف أخطاء البرنامج وتصحيحها صعباً ، إذا لم يكن مستحيلاً. والسبب الآخر لعدم تضمين برامج داخل ملفات الترويسة هو أنها مخالفة لهدف هذه الملفات والتي أوجدت خصيصاً لتحتوي على موجهات المعالج الأولي preprocessor directives. إن موجه #include تضمين ملف ترويسة ليس كاستدعاء ملف برنامج آخر باستخدام أمر DO فعند تضمين ملف ترويسة باستخدام الموجه #include فإن المعالج الأولي سيستخدم ما يحتاجه فقط وكنتيجة لذلك فسيصبح حجم برنامجك الذى يتم تجميعه ضمن أصغر حد ممكن.

ويمكنك عمل تداخل لأمر التضمين في أمر آخر حتى تصل إلى ١٦ مستوى ، على النحو التالي:

```
* FILE 1 . PRG
#include  "file2.ch"
.
*FILE2.CH
#include  "file3.ch"
.
* FILE3.CH
#include  "file4.ch"
```

ملفات ترويسة كليبر 5.x

لقد تم تزويد الإصدار 5.x من كليبر بالملفات التالية ، والتي ستجدها داخل دليل \CLIPPER5\INCLUDE. ويحتوي كل ملف ، ما عدا ملف STD.CH على ثوابت بيان تتبع تسمية اصطلاحية ثابتة بحيث تصبح سهلة الحفظ وهي على النحو التالي:

اسم الملف	يتعلق بـــ:	Prefix السابقة
ACHOICE.CH	الوظيفة ACHOICE.CH	AC_
BOX.CH	أوامر رسم المربع	B
COMMON.CH	عدد مفيد من الوظائف الزائفة	n7a
DBEDIT.CH	الوظيفة DBEDIT( )	DE_
DBSTRUCT.CH	الوظيفة DBSTRUCT( )	DBS_
DIRECTRY.CH	الوظيفة DIRECTORY( )	F_
ERROR.CH	شفرات الخطأ في كليبر 5.x	EG_
FILEIO.CH	الوظائف الدنيا للملف	F_,FS_,FO_,FC_
INKEY.CH	قيم الوظيفة INKEY( )	K_
MEMOEDIT.CH	الوظيفة MEMOEDIT( )	ME_
SET.CH	الوظيفة SET( )	_SET_
SETCURS.CH	الوظيفة SETCURSOR( )	SC_
STD.CH	تعريفات اللغة القياسية	n/ a

وبدلاً من بيان محتويات كل من هذه الملفات على حدة سنقدم مثالاً عملياً يشرح كلاً منها بحيث نستخدم عدداً منها في آن واحد. (يرجى الانتباه إلى أن القرص المرافق لهذا الجزء من الكتاب يتضمن ملفاً معدلاً باسم INKEY.CH والذي يحتوي على مفاتيح إضافية تعيد قيم مفتاح INKEY()). وهي على النحو التالي:

```
#include   "box.ch"
#include   "inkey.ch"
#include   "set.ch"
#include   "setcurs.ch"
#define  OFF  .F.
#translate  CENTER( <row> , <msg> [ ,  <width> ] )  =>  ;
    setpos( <row> ,  int( (  IF  ( len  ( # <width> ) = = o ,     ;
    maxcol( ) + 1,  val( # <width> ) )  -  len( <msg> ) )   /  2 ) )   ;
    ;  disput( <msg> )

function  main
local  key ,  oldcursor  :=  setcursor(SC_NONE)  //  turn  off  cursor
set( SET_SCOREBOARSD ,  OFF )
```

```
set( SET_CANCEL , OFF )
@ 0 ,0, 24 , 79 , box B_DOUBLE + ' ' color 'w/b'
@ 6 , 6 , 18 , 73 , box B_SINGLE + ' ' color 'w/b'
@ 11 , 18 , 13 , 61 BOX B_SINGLE_DOUBLE + ' ' COLOR '+W/RB'
do while key != K_ESC
    center(12 , " press a key - Esc to exit " )
    key := inkey(0)
    scroll(12 , 19 , 12 , 60 , 0 )
    do case
        case key == K_ENTER
            center( 12 , " You pressed Enter " )
        case key == K_F1
            center( 12 , " No help available " )
        case key == K_SH_F1
            center( 12 , " still no help available " )
        case key== K_ALT_A
            center( 12 , " You pressed ALT-A " )
        case key == K_CTRL_Y
            center( 12 , " You pressed Ctrl-Y " )
        otherwise
            center(12 , " Unknown keypress " )
  endcase
  inkey(1)
enddo
setcursor(oldcursor) // restore cursor
```

تجنب تكرار الإعلانات

إذا حاولت تعريف "ثابت بيان" قد سبق تعريفة باستخدام أمر #Define سيصدر لك المعالج الأولي تحذيراً. فمثلاً: إذا قمت بتضمين الملف INKEY.CH داخل ملف الترويسة الخاص بك (وليكن هذا الملف هو MYHEADER.CH) ، ثم ضمنت كلا من هذين الملفين في برنامجك فستحصل على تحذير لكل توجيه موجود في ملف الترويسة INKEY.CH ، فيرجى الانتباه.

```
#Include "myheader.ch" // which #Includes "inkey.ch"
#Include "inkey.ch"
//stand back and watch thise warning fly !
```

وهناك طريقة بسيطة لتجنب هذه التحذيرات وذلك بكتابة أمر #Include متداخلة باستخدام الموجه #ifendf والذي يختبر ما إذا تم تعريف بعض الأمور في ملف الترويسة المناسب. وعلى سبيل المثال:

```
#ifndef K_ENTER                  // defined in INKEY.CH
   #include  "inkey.ch"
#endif
```

وسيتم تضمين ملف الترويسة المسمى INKEY.CH فقط إذا لم يكن موجودا في الملف myheader.ch .

وهذا المنطق مستخدم داخل ملف الترويسة القياسي STD.CH الذي يتم تزويده مع كليبر بحيث يضمن استدعاء "ثوابت البيان" التي تعرف SET( ) في الملفات التالية:

```
 #ifndef   _SET_DEFINED
#include    "set.ch"
#endif
```

وهذا تحسين جديد أدخل على البرنامج حديثاً ، إلا أن هناك احتمالاً أنه لديك عدة ملفات ترويسة تعتمد على ملف ترويسة واحد بعينه. لذلك يستحسن أن تتأكد من سلامة ملفات الترويسة في برنامجك ضد مثل هذه الأخطاء ببنائها على الشكل التالي:

```
#ifndef  _CONSTANTS_EXIST          // see below
   #define  . . .
   #define  . . .
    . . .
   #define  _CONSTANTS_EXIST
#endif
```

حتى إذا كنت ترغب بتضمين ملف الترويسة ذاته أكثر من مرة في برنامجك فيجب تحديد ثوابت بيانه في المرة الأولى فقط. ويستخدم ثابت البيان CONSTANTS_EXIST ليمنع المعالج الأولي من دخول المنطق الشرطي.

الأوامر المعرفة من قبل المستخدم

هناك عدد من المحاذير لاستخدام خيار #define لإنشاء وظائف للمعالج الأولي. وأكبر هذه المحاذير ما ذكرناه أعلاه ، وهو أن المعالج الأولي يعامل توجيهات أمر #define على أنها "حساسة للحالة" ، وهذا يعني أن خطأً واحداً في حرف أو رمز يحول دون ترجمة المعالج الأولي للوظائف الزائفة pseudo-functions بشكل صحيح. أما المحذور الآخر ، فإذا أردت أن تقوم بأي عمل آخر ، غير الترجمة البسيطة للأوامر ، مثلاً: تحويل متغير ما إلى سلسلة حرفية أو كتلة شيفرة ، بحيث يمكن أن تؤدي كل من الأوامر التالية دورها ، مثل: #command و #translate و #xcommand وأخيراً #xtranslate وتمكننا هذه الموجهات والتعليمات من إنشاء الأوامر الخاصة بنا كمستخدمين لكليبر.

إذا أردت مثالاً جيداً عن هذه التوجيهات فارجع إلى ملف الترويسة المسمى STD.CH والذي يحتوي على عشرات من الأوامر المعرفة من قبل المستخدم ، بل لعلك كلما أمعنت النظر في ملف الترويسة هذا شعرت أنه لم يعد هناك "أوامر" ، وسترى أن كل أمر من الأوامر تتم معالجته مسبقاً بواسطة المعالج الأولي داخل استدعاء وظيفة أو أكثر من استدعاء. ولعل هذا الأمر بذاته خبرة مدهشة.

ولقد تم تزويد ملف الترويسة المسمى STD.CH بهدف المراجعة فقط ، وقد تم وضع محتوياته داخل مجمِّع كليبر (CLIPPER.EXE) لهدف الأداء. أما إذا أردت تعديل أي من أوامر كليبر القياسية الثابتة ، فإننا نقترح أن تتأكد من عمل نسخة احتياطية من ملف الترويسة STD.CH ، ثم تأكد بعد ذلك من أنك تعدِّل الملف الجديد فقط باستخدام الخيار U/ مثلاً :

```
C:\>clipper myfile /uMYSTD.CH
```

وسترى رسالة مفادها "تحميل التعريفات القياسية من ملف الترويسة المعدل" "Loading Standard Defs from MYSTD.CH" وهذا يشير إلى أن المجمّع يحضر مجموعة الأوامر من الملف الذي حددّته له ليتجاهل الأوامر المفترضة في البرنامج الأصلي.

أما إذا أردت تعديل عدة أوامر فقط ، فيمكن أن تضعها في ملف ترويسة خاص ثم تضمنه باستخدام الخيار include# في برنامجك ، وسنبين هذا المبدأ لاحقاً في هذا الفصل.

أما الشكل الأساسي لعبارة الأوامر التي يعدّها المستخدم فهي على النحو التالي:

```
#command  ( or #translate)   <input text>  =>  <result  text>
```

تحتوي صيغة الأمر التي يعدّها المستخدم على ثلاثة أجزاء أساسية وهي : نص الإدخال ، فاصل السحب ("=>") ، ثم نص الناتج أو الإخراج.

وهناك تمييز هام بين كل من الموجه command# الموجه translate# وهو أن الأوامر التي يعدّها المستخدم باستخدام الموجه translate# يمكن أن تظهر في أي مكان في العبارة (كما هو الحال في المعرف defines#). وعلى النقيض من ذلك ، فإن الأوامر التي يعدّها المستخدم باستخدام الموجه command# يجب أن تكون في أول حرف على السطر دون أن يسبقها أية مسافة فارغة. فعلى سبيل المثال ، انظر إلي إعادة تعريف كليبر للأمر **CLEAR** "امسح":

```
#command  CLEAR   =>   __clear( )  ;  __KillRead( )  ;  GetList  := { }
```

فإذا حاولت استخدام هذا الأمر على النحو التالي:

```
x  := clear
```

فلن يكون المعالج الأولي قادرا على ترجمته على أنه أمر ، وكذلك سيظن المجمّع أن أمر CLEAR هو "متغير" أو "حقل". إلا أنك إذا إستعملت الموجه translate# على النحو التالي:

```
#translate CLEAR  =>  __clear( ) ;  __KillRead( ) ; GetList := { }
```

فيمكن عندئذٍ استخدام أمر CLEAR في أي مكان في هذه العبارة.

نص الإدخال Input text

هذا هو الأمر الذي يبحث عنه المعالج الأولي أثناء مسح شيفرة المصدر الخاصة بك. ويمكن أن يحتوي نص الإدخال على واحد من الأمور التالية أو جميعها ، وهي:

- قيم حرفية **Literal values** : وهي الحروف التي يجب أن تظهر كما هي تماماً في نص الإدخال بحيث يمكن أن يترجمها المعالج الأولي. ومثال على القيم الحرفية هو "@" في أمر : CLEAR .. @ :

```
#command @ <top> , <left> CLEAR  =>  ;
              Scroll(  <top> , <left> )  ;     ;
              Setpos(  <top> , <left>  )
```

- كلمات **words** : هي كلمات هامة وأساسية تتم مطابقتها طبقاً لمبدأ ما يسمى : "عادة أحترام الوقت في dBASE قاعدة البيانات" (وهذه الكلمات غير حساسة للحالة ، وتؤخذ منها الحروف الأربعة الأولى فقط). فإذا كتبت مايلي:

```
@ 0 , 0 clea
```

فإن المعالج الأولي سيبقى قادراً على ترجمتها وفقاً للموجه #command المعطى لأمر: CLEAR..@.

- قابل العلامات **Match-Marker** : وهي "المتغيرات" التي تختلف طبقاً للأمر الذي يحدده المستخدم. وتتم معاملة هذه المتغيرات بشكل يختلف عن معاملة عبارات

التعريف #define ، بحيث يتم هناك تعريف المتغيرات ببساطة بين قوسين على النحو التالي:

```
#define TIMES( a , b )  (a)  *  (b)
```

إلا أنك عند استخدام أي من الموجهين #translate أو #command يجب أن تحيط مثل هذه المتغيرات بإشارتي "<" و ">" عند بدايتها ونهايتها على الشكل التالي:

```
#translate TIMES( <a> , <b> )  =>  ( <a> ) * ( <b> )
```

ويحدد أمر "قابل العلامات Match-markers" اسماً لكل متغير ، ويمكنك الإشارة اليه بعد ذلك في الإخراج (أو نص "الناتج result"). وفي مثال الضرب TIMES( ) المبين أعلاه ، كيف أن "قابل العلامات" يعلّم ويعيين جزأين من النص هما: a و b.

ويتطابق "قابل العلامات" مع "معلّمات النتيجة" ، والذي يكتب النص الناتج عن ترجمة المعالج الأولي. ويمكن بسهولة أن ترى من مثال TIMES( ) كيف تم تشكيل كل من <a> و <b> لتظهر في ناتج المعالج الأولي لهذا الأمر. (وسنبين لاحقاً خيار "معلمات النتيجة" بمزيد من التفاصيل).

وسيتم تطبيق المصطلحات الجديدة التالية أثناء مناقشة "قابل العلامات match-marker":

- **"Stringify"** (ضع على شكل سلسلة): حوّله ليصبح على شكل سلسلة حرفية.
- **"blockify"** (ضع على شكل كتلة): حوّله ليصبح على شكل كتلة.
- **"Logify"** (ضع على شكل منطق): حوّله ليصبح على شكل قيمة منطقية.

الأمران التوجيهيان #xtranslate و #xcommand

هذان الأمران مماثلان تماماً لكل من أمري **#translate** و **#command** ، باستثناء شيء واحد ، وهي أنهما يتطلبان مقابلة تامة. أما أمر **#command** و **#translate** فيحملان معهما إرثاً من الخزي والعار ينسب إلى التوافقية مع قاعدة البيانات dBASE. ولا يتطلبان سوى مطابقة الحروف الأربعة الأولى فقط من نص الإدخال. وسبب هذا النقص هو أن مفسِّر قاعدة البيانات **+dBASE III** سمح للمبرمجين اختصار الأوامر واستخدام الحروف الأربعة الأولى منها.

ويعتبر هذان الأمران التوجيهيان #xtranslate و #xcommand منقذين للحياة في الحالات التي تتطلب إجراء دوائر. فلنفترض أنك تريد تنفيذ وظيفة باستخدام المعالج الأولي تسمى ( )Dateword والتي ترجع التاريخ الفعلي للنظام:

```
#translate DateWord( )  =>  ;
              cmonth( date( ) ) + ' ' + ;
              ltrim( str(day(date( ) ) ) )  + ',' + ;
              str( Year( date( ) ) ,  4 )
function main
? dateword( )
return nil
```

فإذا حاولت تجميع هذا البرنامج فستحصل على رسالة "خطأ قاتل":

```
Fatal error. Input buffer overflow
```

(وتكفي كلمة "قاتل" أن تدب الهلع في قلب المبرمج !). ويحدث هذا بسبب الأمر التوجيهي الصادر عن أمر **#translate** إذ ينظر المعالج الأولي على الحروف الأربعة الأولى فقط ، وبذلك يخطيء بتفسير أمر **deteWord( )** على أنه date فقط ، آخذاً الحروف الأربعة الأولى من الكلمة فقط بعين الاعتبار وهو في الوقت ذاته استدعاء لوظيفة أخرى هي ()Date. وكما ترى ، فإن هناك عدة مرات ذكر فيها أمر **date( )** في ناتج النص وهذا يسبب دوائر غير قابلة الاسترجاع.

ويستحسن أن تستخدم مستقبلاً الموجهين **#xtranslate** و **#xcommand** أما الموجهان **#translate** و **#command** فهما مفيدان فقط عند التوافقية مع مختصرات أوامر قاعدة البيانات dBase. أما إذا لم ترغب اختصار أوامرك (ولاداعي لذلك في بيئة تجميع على غرار كليبر) فلن يكون هناك أي داعٍ لاستخدام الموجهين **#translate** و **#command**.

علامات المقابلة Match-Markers

لعل من أصعب مفاهيم كليبر فهماً "علامات المقابلة" (ومعلّمات النتيجة). فهناك العديد من أنواع "علامات المقابلة" كل منها يلبي غاية محددة. فإذا لم يتعرف المبرمج على كل من هذه العلامات ، ويدرك دورها المتميز عن غيرها ، فلا داعي أن تنفق الوقت قلقاً عنها واهتماماً بها وبتفسيراتها. وأما مايجب أن تركز عليه وتعتمد عليه باستمرار فهو "علامات المقابلة" العادية فقط. فإذا أصبحت متمرساً تماماً بالتعامل مع المعالج الأولي وأصبحت ذا خبرة واسعة باستخدام الأوامر المعرفة من قبل المستخدم user-defined commands ، عندئذ فقط يمكنك العودة إلى هذا القسم وتعلّم كيف وأين يمكنك استخدام "علامات المقابلة" المتخصصة.

القاعدة اللغوية	النـــــوع
<name>	علامات المقابلة (العادية)
<name, ...>	قائمة علامات المقابلة
<name : word list>	علامات المقابلة المقيدة
<*name*>	علامات المقابلة الشاذة
<(name)>	علامات المقابلة

علامات المقابلة العادية Regular match-marker

يعتبر هذا الخيار أشهر علامات المقابلة على الإطلاق. ويقوم بكل بساطة بمقابلة التعبير الصحيح legal expression التالي في نص الإدخال. ويستخدم غالباً مع "معلم النتيجة" العادي إلا أنه يمكن استخدامه مع الموجه "Stringify" (ضع على شكل سلسلة حرفية) ، و "blockify" (ضع على شكل كتلة برامج). وخير مثال على هذا النوع من علامات المقابلة هو أمر Do While :

```
#command Do WHILE <exp>  => while <exp>
```

فإن كل ماتحدده على أنه "تعبير" <exp> سينسخ كما هو تماماً إلى نص الإخراج.

قائمة علامات المقابلة List match-marker

يمكن هذا الأمر المعالج الأولي من مقابلة قائمة التعابير التي تفصل عن بعضها بفاصلة. فإذا لم يطابق نص إدخال علامة مقابلة ، فلن يحتوي اسم العلامة المحددة على أي شيء وبالتالي، فإنه لن يستخدم في نص النتيجة. وخير مثال على قائمة علامات المقابلة هو الأمر ? ، والذي يقبل قائمة اختيارية من المتغيرات. وإذا لم تحدد المتغير الإضافي فإن أمر ( )QOUT سيضع رمز الرجوع carriage return وسطر التغذية line feed كنتيجة لهذا بكل بساطة.

```
#command  ?  [ <list, . . . > ]  =>  QOUT( <list> )
```

علامات المقابلة المحدودة Restricted match-marker

يستخدم هذا الأمر لمعالجة نص إدخال يجب أن يطابق كلمة واحدة في قائمة تم تفريقها عن بعضها باستخدام الفواصل. فإذا لم يكن نص الإدخال موجوداً في القائمة المحددة فستفشل عملية المطابقة ولن يحتوي اسم العلامة أي شيء.

يستخدم هذا النوع من أعمال المطابق غالباً عند استخدام معلّمات النتيجة المنطقية logify result-marker لكتابة قيمة منطقية في نص ناتج. ويبين المثال التالي كيفية عمل هذا الخيار:

```
#command DRAW BOX [ <double : DOUBLE> ]  =>  draw_box( <.double.> )
```

إذا حددت العبارة أو الفقرة DOUBLE الاختيارية ، فسيبدو ناتج النص كما يلي:

```
draw_box ( . t . )
```

ويمكنك بعد ذلك إنشاء وظائفك بحيث تختبر القيمة المنطقية وتعمل بناء عليها.

علامات المقابلة العشوائية Wild match-marke

يقوم هذا الخيار بمقابلة نص الإدخال من الموقع الحالي إلى نهاية العبارة ، ويستخدم عادة لمقابلة نص إدخال قد لايكون نصاً صحيحا. ويمكن إعطاء مثال ملحوظ عن استخدام هذا الخيار في قسم "التوافقية" في ملف الترويسة STD.CH والذي يعنون مختلف الأوامر المطلقة في قاعدة البيانات dBASE III :

```
#command SET  ECHO <*x*>        =>
#command SET  HEADING  <*x*>    =>
#command SET  MENU  <*x*>       =>
#command SET  STATUS <*x*>      =>
#command SET  STEP  <*x*>       =>
#command SET  SAFETY  <*x*>     =>
#command SET  TALK  <*x*>       =>
```

قد يبدو هذا للوهلة الأولى وكأنه نكتة ، إلا أنه في الواقع طريقة مفيدة للتعامل مع كليبر ، ولايستطيع كليبر أن يقبل التعامل مع أي من هذه الأوامر ، لذلك فيجب على المعالج الأولي محاولة العثور عليها وشطبها من البرنامج الذي يراد تجميعه.

فإذا أدخلت السطر التالي في شيفرة المصدر الخاصة بك:

```
set echo, is there an echo in here ?
```

فسيخرج المعالج الأولي سطراً فارغاً دون تردد.

ويمكن استخدام علامات المقابلة العشوائية أيضاً لتجميع نص الإدخال في نهاية العبارة وكتابتها في نص الناتج باستخدام أحد "معلّمات النتيجة المسلسلة" stringify result-marker. وإن أفضل مثال على استخدام هذا الأمر هو عبارات END المختلفة. ويرغب بعض المبرمجين توثيق برامجهم لتدل بوضوح على نهاية كتل الشيفرة كما هو مبين فيما يلي:

```
do while condition1
   *
enddo condition1
```

وستسبب الكلمة الإضافية على سطر ENDDO بعض المشاكل للمجمّع عندما لاتكون في محددة في تعريف العبارة ENDDO في ملف الترويسة STD.CH.

```
#command ENDDO  <*x*>      => enddo
```

حيث يعمل هذا النوع على نزع أية كلمة متعبة بحيث يمكنك إبقاء تعليقاتك الإضافية ويمكن للمجمع أن يتابع عمله بشكل طبيعي.

التعبير الموسع لعلامات المقابلة Extended expression match marker -

سيطابق هذا الأمر كلا من التعبير العادي أو الموسع بما في ذلك اسم الملف أو مواصفات المسار path. ويمكنك هذا من تمرير التحديد دون علامات تنصيص ، أو دون أقواس كما هي الحال في التعبير الموسع. ثم يمكنك بعدئذ استخدام "معلم النتيجة المسلسل" الذكي

stringify result-marker بحيث تضمن أن التعابير الموسعة لم تتسلسل. ويُعطي أمر مجموعة الافتراضات SET DEFAULT مثالاً على التعبير الموسع لمعلم النتيجة.

```
#command SET DEFAULT TO <(path)> => set(_SET_DEFAULT, <(path)>)
SET DEFAULT TO c:\aquarium    // Set(_SET_DEFAULT, "c:\aquarium" )
SET DEFAULT TO ( "c:\aquarium" ) // Set(_SET_DEFAULT, ("c:\aquarium") )
```

العبارات الاختيارية Optional Clauses

يمكنك تحديد عبارات اختيارية للمطابقة بوضعها داخل أقواس كبيرة معقوفة " [ ] " ويمكن أن تحتوي هذه العبارات قيماً حرفية ، أو كلمات ، أو معلمات نتائج ، وغيرها من العبارات الاختيارية. وهناك نوعان من هذه العبارات هما:

- كلمة رئيسة متبوعة بمعلّم نتيجة ، مثال : @..GET

```
#command @ <row> , GET <var> [ PICTURE <pic> ] ...
```

- كلمة رئيسة بذاتها مثل : SET NKEY TO :

```
#command SET KEY <n> [ TO ] => Setkey( <n> , NIL )
```

نص الناتج Result Text

إن نص الناتج هو من مخرجات المعالج الأولي بعد ترجمة البرنامج. ويمكن أن يحتوي هذا النص على أي من العناصر الثلاثة التالية ، أو جميعها:

- قيم حرفية *Literal values* : وهي حروف تكتب مباشرة إلى نص الناتج. وتوجد أمثلة على القيمة الحرفية في كل أمر معروف من قبل المستخدم تقريبا.

- كلمات *Words* : وهي كلمات أو معرّفات تكتب مباشرة إلى نص الناتج وهي كمثيلتها أعلاه. ويوجد مثال على هذه في كل أمر معرف من قبل المستخدم تقريبا.
- معلّمات نتائج *Result-Marker* : تشير هذه المعلّمات إلى اسماء المعلمات ذاتها. وكما أشرنا سابقاً تتم كتابة نصوص الإدخال التي تتم مطابقتها إلى نص الناتج على أنها "معلّمة ناتج" وكما هو الحال في علامات المطابقة ، فإن معلمات النتيجة يجب أن تحاط بإشارتي "<" و ">".

معلمات الناتج Result-marker

هناك العديد من معلمات الناتج ، كما هي الحال مع علامات المطابقة. ولاتقلق الآن إذا لم تفهم عمل كل منها بالتحديد. وأما التي ستستخدمها غالباً فهي معلمة الناتج العادية. وكلما تعرفت على كتابة الأوامر المعرفة من قبل المستخدم المتقدمة ، يمكنك الرجوع إلى هذا القسم لتتعلم أين وكيف تستخدم هذه المعلّمات المتخصصة.

النـــوع	القاعدة اللغوية
معلم النتيجة العادي Regular result-marker	<name>
معلم نتيجة سلسلة صامتة Dumb stringify result-marker	#<name>
معلم نتيجة سلسلة عادية Normal stringify result-marker	<"name">
معلم نتيجة سلسلة ذكية Smart stringify result-marker	<(name)>
معلم نتيجة كتلة Blockify result-marker	<{name}>
معلم نتيجة منطقي Logify result-marker	<.name.>

معلم الناتج العادي Regular result-marker

يكتب هذا الخيار نص الإدخال المطابق في نص الناتج. ولن يكتب شيئاً إذا لم يجد ما يقابله. ويعتبر هذا الخيار أشهر معلّمات الناتج ، وبالتالي فيحتمل أن يكون أكثرها استعمالاً ، كما هي الحال مع علامات المطابقة العادية. ويستخدم هذا الخيار غالباً مع علامات المطابقة العادية إلا أنه يمكن استخدامه مع أي منها على الإطلاق. وتبين الوظيفة الزائفة TIMES( ) مثالاً سريعاً على معلم الناتج العادي.

```
#xtranslate  TIMES(  <a> ,  <b>  =>  ( <a> )  *  ( <b> )
```

معلّم ناتج سلسلة صامتة Dumb stringify result-marker

يحول هذا الأمر نص الإدخال المطابق إلى سلسلة حرفية ويكتبها في نص النتيجة. فإذا لم يطابق أي نص إدخال فسيكتب المعالج الأولي سلسلة حرفية قيمتها صفر ، أي فارغة " ") (في نص الناتج. أما إذا استخدم مع قائمة علامات المقابلة ، فستحول القائمة إلى سلسلة حرفية وستكتب في نص الناتج. وخير مثال على معلمات الناتج هو أمر SET COLOR TO والذي يقبل مواصفات لون غير مدرج ، على النحو التالي:

```
#command SET COLOR TO  [ <*spec*> ]  =>  SetColor( #<spec> )
set  color to  w/b                          //    SetColor(  " w/b " )
```

معلم ناتج المتسلسل العادي Normal stringify result-marker

يشبه هذا المعلم سابقه إلى حد كبير. وهناك اختلافات بسيطة بينهما وهي أنه إذا لم يمكن مطابقة أي نص إدخال فإن هذا المعلّم لن يكتب شيئاً (بدلاً من سلسلة فارغة). وكذلك إذا استخدم هذا المعلّم مع قائمة مطابقة ، فسيكون كل عنصر في القائمة على شكل متسلسل بدلاً من تحويل القائمة كلها ككل. ويعطي أمر RELEASE مثالاً جيداً على معلّم الناتج المتسلسل العادي:

```
#command   RELEASE <vars, . . .>   =>  __MXRelease(   <" vars">   )

release  mvar                          / /  __MXRelease(   "mvar"   )
release  mvar ,  mvar2 ,  mvar3  / / __MXRelease(  "mvar" ,  "mvar2" , "mvar3"  )
```

وبالمقارنة. فإن هذا سيحدث إذا استخدمت قائمة باستخدام أمر معلم الناتج المسلسل الصامت.

```
#command   RELEASE  <vars, . . .>   => __MXRelease( # <vars>   )

release  mvar ,  mvar2 ,  mvar3   / / __MXRelease(  "mvar" ,  "mvar2" , "mvar3"  )
```

معلِّم الناتج المتسلسل الذكي Smart stringify result-marker

يحول هذا الأمر النص المطابق إلى سلسلة حرفية فقط إذا لم يكن موضوعاً داخل قوسين. أما إذا لم يُطابق نص إدخال ، فلن يكتب شيء في نص الناتج. أما إذا استخدم هذا الأمر مع أمر قائمة معلم الناتج ، فتتم سلسلة كل عنصر في القائمة باستخدام القاعدة ذاتها وتكتب في نص الناتج.

وقد صمم هذا الأمر لدعم التعابير الموسعة بشكل خاص للأوامر التي تختلف عن مجموعات SETs. وأحد هذه الاستخدامات هو أمر EREASE :

```
#command   ERASE   < ( file ) >            =>  FErase(   < ( file ) >  )
mvar  :=  "temp.dbf "
 ERASE  temp.dbf                              //   FErase(   "temp . dbf "   )
 ERASE  ( mvar )                              //   FErase(  ( mvar )   )
```

معلم الناتج الكتلي Blockify result-marker

يحول هذا الأمر نص الإدخال المطابق إلى كتلة شيفرة code block. أما إذا لم يطابق نص إدخال فلن يكتب أي شيء في نص الناتج. وأما إذا استخدم هذا الأمر مع قائمة معلّم الناتج فيتم تحويل كل عنصر في القائمة. وتعتمد كثير من الوظائف البرمجية في كليبر 5.x

على كتل الشيفرة code blocks ، بحيث يصبح هذا الأمر هاماً جداً. ويبين أمر SET FILTER مثالاً على التكتيل:

```
#command SET FILTER TO <xpr>  => dbSetFiler(  <{xpr}> , <"xpr"> )
set filter to ! deleted( )  =>  dbSetFiler(  { | | ! deleted( ) } , ;
                                              " ! deleted( ) " )
```

معلم الناتج المنطقي Logify result-marker

يكتب هذا الأمر عبارة "حقيقي" True (.T.) إلى نص الناتج إذا تمت مطابقة نص الإدخال ، أو عبارة "غير حقيقي" False (.F.) إذا لم تتم المطابقة ، ولن يُكتب نص الإدخال ذاته في نص الناتج. وكما أشرنا آنفاً ، فإن أفضل استخدام لهذا الأمر هو عندما نستخدم معلم الناتج المحدد. ونبين فيما يلي مثالاً على هذا في أمر SET MESSAGE TO :

```
#command  SET MESSAGE TO <n> [<cent: CENTER, CENTER>]  =>  ;
          SET(_SET_MESSAGE , <n> ) ; SET(_SET_MCENTER, <.cent.> )

set mesage to 24                 // set(_SET_MESSAGE ,  24 ) ;
                                 // set( _SET_ MESSAGE ,  . F . )
set mesage to 24 center          // set( _SET_ MESSAGE ,  24 ) ;
                                 // set( _SET_ MESSAGE ,  . T . )
```

سطور المتابعة Continuation lines

يمكن أن يحتوي نص الناتج ، كما رأيت من الأمثلة السابقة ، أكثر من عبارة واحدة . ويجب تفريق كل عبارة عن التي تليها باستخدام فاصلة منقوطة " ; ". ولدى البدء باستخدام أوامر من إعدادك أكثر تعقيداً تحتاج إلى سطور متابعة ، يجب أن تتأكد من وجود الفاصلة المنقوطة " ; ".

الرموز المحجوزة Reserved Characters

إذا أردت استخدام إشارة " أصغر من "< " أو القوس المعقوف الأيسر "]" في نص الناتج فيجب أن تسبقها بشرطة مائلة معكوسة للخلف " \ " ، ويعتبر هذا الأمر ضرورياً لأن كلاً من هذه الرموز يحمل معنى خاصاً بها فيما يتعلق المعالج الأولي. فإن إشارة "< " مثلاً تعني بداية علامة مطابقة ، كما أن الأقواس المعقوفة تعني عبارات أو فقرات اختيارية.

لذلك ، يجب توخي الحذر والدقة عند كتابة نص الناتج مثل هذه الرموز التي تتطلب وجود أقواس بشكل طبيعي. فإذا نسيت استخدام الشرطة المائلة المعكوسة فسيتم حذف إسنادات تمييز المصفوفة تماماً ، وسينجم عن هذا إزعاج كبير لك لدى تشغيل برنامج Debugger.

وتجدر الإشارة إلى أنه يمكنك استخدام الفراغات بالشكل الذي تريده وبحرية تامة في كل من أمري **#xcommand** و **#xtranslate** ، ويحتاج المعالج الأولي إلى هذه الفراغات بحيث يمكنه تحويل كل شيء مناسب. ولعل المكان الوحيد الذي لاتحتاج فيه إلى فراغ هو مابين القوس الزاوي ومعلّمات الناتج (مثال: استخدم "<msg>" بدلاً من "< msg >") . وسيتم التجميع بشكل صحيح وسليم ، إلا أن قراءة شيفرة المصدر ستكون أصعب وأبطأ قليلاً إذا وجد الفراغ.

إن إحدى فوائد كتابة الأوامر المعرفة من قبل المستخدم ذاتياً هو الحدّ من تضارب الاسماء بين الوظائف التي تريدها ، والوظائف الأخرى التي تحمل الاسم ذاته. ويعتبر أمر CENTER( ) خير مثال على هذه الحالة ، حيث أن كل مبرمج يستخدم هذا الأمر تقريباً، ويستخدم كل من المبرمجين تركيبة لغوية تختلف فيما بينهم واحداً عن الآخر. وقد يحدث هذا مشاكل كبيرة عند استدعاء الوظيفة CENTER( ) الذي يكون موصولاً بغيره من الأوامر. وقد تحصل على أخطاء عدم مطابقة ""type mismatch errors" أثناء التشغيل ولا تعرف سبباً لهذه الأخطاء.

إلا أنك إذا حولت الوظيفة CENTER( ) إلى أمر يعرّفه المستخدم user-defined command ، كما فعلنا آنفاً فإن هذا الأمر لن يعود موجودا على أنه "وظيفة". ويتم

التوسيط مباشرة بحيث يلغي أي تعارض اسماء محتمل. ولن يكون هناك داعٍ للقلق بعدئذ أن يقوم هذا الأمر باستدعاء وظيفة ( )CENTER أخرى إذا اتبعت هذا المثال.

ولـدى كتابـة أمـر يعرفّه المسـتخدم باسـتخدام الموجهـين **#command** أو **#translate** سيبقي مرئياً من ذاك السطر وإلى نهاية ملف البرنامج **PRG**. ولن يكون مرئيـاً في ملفـات برامج أخرى. ويبين الجزء التالي من البرنامج هذا المبدأ :

```
/* MAIN.PRG */
#xcommand REDRAW => @ 0, 0, maxrow(), maxcol() box ;
                                      replicate(chr(176), 9)

function main
redraw
do test
return nil

*  eof :  main.prg

/* TEST.PRG */
redraw
return nil

* eof : test.prg
```

ولن يستطيع المعالج الأولي ترجمة أمر REDRAW في ملف TEST.PRG وسيؤدي هـذا الخطأ إلى خطأ أثناء التجميع على الشكل التالي "statement unterminated").

وأما الإستثناء الوحيد لهذه القاعدة هو الأوامر التي يتم تعريفها من قبل المستخدم في ملـف الترويسة STD.CH (أو أي ملف ترويسة قياسي يتـم تحديـده باسـتخدام خيـار التجميـع U/).

الأولوية Precedence

عدة موجهات لكل عبارة

يترجم المعالج الأولي الموجهات الأساسية الثلاثة بالترتيب التالي: **#define** و **#translate** (أو **#xtranslate**) و **#command** (أو **#xcommand**). ويترجم المعالج الأولي كل موجه لدى التعرض له ، ثم يعيد مسح ذاك السطر من الشيفرة للبحث عن أية موجهات أخرى. تأمل الموجهات التالية:

```
#define OFFSET              20
#define FROMBACK(a)    len( a - OFFSET )
#xtranslate addem( <a> ) =>  ;
            aeval( <a> , { |ele |  msum += ele } , FROMBCK(  <a>  ) )

addem( myarray )
```

وعندما يصادف المعالج الأولي للأمر الذي عرّفه المستخدم AddEm( ) فإنه سيحوّله إلى مايلي:

```
aeval( myarray , { | ele |  msum += ele } , FROMBCK( myarray ) )
```

ثم يمر على ذلك السطر مرة ثانية ، وسيكتشف FROMBACK ، إذ أنه قد تم تعريفه بواسطة الموجه #define على أنه وظيفة برمجية زائفة ، وستتم ترجمته هو الآخر أيضاً على النحو التالي:

```
aeval( myarray , { | ele |  msum += ele } , len( myarray - OFFSET ) )
```

وأخيراً سيعمل المعالج الأولي على تعريف مايسمى OFFSET بحيث يترجمها هي الأخرى على النحو التالي:

```
aeval( myarray , { | ele |  msum += ele } , len( myarray  -  20 ) )
```

وبما أنه لم يبق هناك أية موجهات لمعالجتها في هذا السطر ، فسيعتبر المعالج الأولي أنه أنجز مهمته بنجاح ، وينتقل إلى تنفيذ أعماله الأخرى.

التعريفات الحديثة

سيأخذ المعالج الأولي أحدث التعريفات لكل توجيه directive عند ترجمة الشيفرة.

وهذا يعني مثلاً ، أنك إذا عرّفت ثابت بيان في شيفرة المصدر الخاصة بك ، ثم ضمنته في ملف ترويسة يعيد تعريفه من جديد فإن هذا التعريف سيستخدم ، وستحصل على إنذار تجميعي.

ويبين المثال التالي هذه الحالة:

```
// TEMP.PRG
#define ELEMENTS 5
#xtranslete Center( <a> ) = > space( int( ( 80 - len( <a> ) ) /2) )
#xcommand READ => readmodal( getlist ) ; aadd(mastergets , getlist )
#include "mystuff.ch"
function main
local getlist := { }
local a[ELEMENTS] , mastergets := { } , string := space(40)
scroll( )
@ 2 , 20 get string
read
string = trim(string)
@ 3 , center(string) say string
return nil
* end of file TEMP.PRG

// MYSTUFF.CH
#define ELEMENTS 100
#xtranslate Center( <row> , <msg> ) => ;
            @ <row> , space(int( ( 80 - len( <msg> ) ) / 2 ) say <msg>
```

أما الموجه READ في ملف TEMP.PRG فإنه سيتجاوز الأمر الافتراضي READ في كليبر. وكذلك فإن الموجه CENTER( ) في ملف MYSTUFF.CH سيتجاوز الموجه Center( ) الموجود ضمن ملف TEMP.PRG وذلك لأنك ضمّنت ذلك الملف بعد الموجه الأول.

لاحظ أيضاً أنه يمكنك تجاوز الأوامر القياسية في كليبر على غرار ما فعلنا بأمر READ
هذا المثال. وأما مجموعة القواعد القياسية لكليبر (كما تراها في ملف الترو
STD.CH) فإنها تحمَّل في بداية عملية التجميع. وطبقاً لقاعدة أحدث تعريف ،
المعالج الأولي سيستخدم التعريف الجديد الذي كتبته أنت لأمر READ.

ملاحظة هامة

عند إعادة تعريف كل من الموجهات التالية **#command** و **translate**
#xcommand و **#xtranslate** كما أشرنا في المثال السابق ، فإنك لن تحصل على
تحذيرات من المجمِّع وإنما تنتج التحذيرات عند إعادة تعريف "ثوابت البيان" manifest
constants فقط.

الموجه #error

لقد تمت إضافة هذا الموجه مع الإصدار 5.01 من كليبر. وإذا واجهت المجمِّع فإنه سيو
عملية التجميع تماماً بحيث تصبح شبه ميتة في مساراتها. ولماذا قد يرغب المبرمج بالإ
على عمل من هذا النوع ؟؟ إن أفضل الأسباب لهذا العمل هو أن الموجه #error يمك
من تحصين شيفرتك source code ضد أي خطأ محتمل في الحالات التي يجب أن تعت
فيها بشكل مطلق وأكيد على بعض ثوابت البيان المحددة.

وهناك حالتان على الأقل ، يكون فيهما هذا الموجه هاماً ، وهما :

أ) إذا كنت تعمل مع مجموعة من المبرمجين ضمن فريق عمل واحد.

ب) إذا افترض برنامجك تمرير موجهات محددة على سطر الأوامر باستخدام مفتاح المج
/D.

ويبين لك المثال التالي كيف يمكنك أن تضمن وجود ثابت بيان ITERATIONS:

```
#ifndef ITERATIONS
#error Missing ITERATIONS-aborting compilation
```

```
#endif
```

ويستخدم هـذا الموجـه بشـكل مركـز في ملـف الترويسـة RESERVED.CH ، والـذي يستخدم لاستثناء تعارض الاسماء مابين وظـائفك وأسمـاء الوظـائف المحجوزة لكليبر ذاتـه. ويرجى الرجوع مباشرة الي ملف الترويسـة RESERVED.CH لمزيـد مـن الأمثلـة علـى هذا الموضوع.

الموجه #stdout

لقد تمت إضافة هذا الموجه لكليبر في الإصدار 5.2 ، وهو يستخدم لتوجيه المجمِّع لإخـراج (كتابة) نـص الإخـراج في وسـيلة الإخـراج القياسـية (وعـادة مـاتكون هـذه الوسـيلة هـي الشاشة) أثناء عملية التجميع. ولاداعي لوضع النص المطلوب داخـل علامـات تنصيـص " ". ويمكـن أن يكـون هـذا الأمـر مفيـداً لإرسـال رسـائل إمـا إلى الشاشـة ، أو إلى ملـف السجلات إذا كنت تستخدم DOS في إعادة توجيه مخرجات المجمِّع.

أهمية ملف PPO. لمخرجات المعالج الأولي

أشرنا سابقاً إلى أن خيـار P/ سينتج ملـف"مخرجـات المعـالج الأولي" PPO. . ويشـبه هـذا الملف شيفرة المصدر بعد أن ينتهـي منهـا المعـالج الأولي. ولقـد تم تزويدهـا كمرجـع فقـط. ويمكننا إعطاء مثال جيد لتوضيح العلاقة مابين كل من ملف PPO. وملـف الهـدف OBJ. هو أن يكون أحدهما بمثابة "الشرنقة" (لدودة القزّ أو الحريـر). والآخـر بمثابـة "الفراشـة" ذاتها. فالفراشة في هذا المثال تمثل ملف "الهدف" OBJ. ولن تكون ذات أي نفع بعد ذلك للشرنقة PPO..

ولكن ، قبل أن تنتهي تماماً من ملف **PPO.** لابـد أن تـدرك أن لـه اسـتعمالات في غايـة الأهمية.

١) التعرف على طريقة العمل الداخلي لكليبر 5.x

إن ملف **PPO.** هو وسيلة تعلّم قيمة جداً إذ يمكنك من الأطلاع الدقيق على كيفية قيام المعالج الأولي بترجمة كل أمر من أوامر كليبر.

ونقترح أن تقوم بطباعة ملف الترويسة STD.CH الذي يحتوي على كافة الموجهات directives التي يستخدمها المعالج الأولي لإنشاء ملف PPO. ، وذلك في محاولة لفهم هذا العمل بشكل جيد.

فإذا أخذت بعين الاعتبار ، أنه عند صدور النسخة 5.01 من كليبر التجريبي والذي إستمر فترة طويلة ، لم تكن هناك أية وثائق شرح تفصيلية عن عمل ملف الترويسة STD.CH. وكان على مستخدمي هذه النسخة التجريبية من البرنامج أن يتعرفوا بأنفسهم على المزايا التي تم تغييرها أو إضافتها إلى البرنامج بدراسة ملف الترويسة المذكور.

٢) اكتشاف أخطاء الشيفرة وتصحيحها

إن "ثوابت البيان" رائعة فيما يتعلق بوضوح قراءتها. إلا أنها قد تسبب مشكلة عند تشغيل برنامج Debugger ، إذ أنها تتحل بشكل تام إلى "ثوابت" أثناء وقت التجميع. ولن تكون هناك أية وسيلة للتعرف على قيمها أثناء تشغيل برنامج اكتشاف الأخطاء وتصحيحها Debugger.

إلا أن كليبر ، ولحسن الحظ ، يمكنك من مشاهدة مخرجات المعالج الأولي إلى جانب شيفرة المصدر التي تعدها. فإذا توفر ملف PPO. للملف البرنامج الحالي PRG. ، فسيعرض برنامج Debugger رقم كل سطر مقابل للسطر الذي يعمل عليه. ونبين فيما يلي مثالاً صغيراً على مايمكنك ، أن تشاهده (وقد تمت كتابة البرنامج بحرف غامق ، بينما كتبت مخرجات المعالج الأولي بحرف عادي لتمييزها عنها. ولنفترض أن ثابت البيان FNAME قد تم تعريفه سابقاً باسم "Sara".

```
if lastkey( )  != K_ESC
if lastkey( )  <> 27
   use customer  new
   dbUseArea( .T. , , "customer " , , if ( .F.  .OR.  .F.  ,  !  .F. , NIL ) ,  .F.  )
   set  index  to  customer
   dbClearIndex( ) ;  dbSetIndex(  "customer"  )
  seek  FNAME
  dbSeek (   "Sara"   )
endif
endif
```

ويمكنك هنا أن تلحظ ان هذا أكثر وضوحاً من أن تحاول تنفيذ برنامج Debugger دون مخرجات المعالج الأولي.

٣) تحسين برامجك إلى أقصى حد

تتم ترجمة معظم أوامر كليبر من قبل المعالج الأولي إلى أكثر من استدعاء وظيفة واحد ، ولعلك لن تحتاج في كثير من الأحيان إلى استخدام هذه الاستدعاءات كلها. ولعل أفضل مثال على هذا هو استخدام أمر SAY . .@ لإعادة موقع المؤشر إلى ماكان عليه سابقاً أما في كليبر Summer'87 فلقد كان الخيار الوحيد هو:

```
oldrow   := row( )
oldcol    :=  col( )
*
@  oldrow ,  oldcol  say  ' '
```

إلا أن هذا ستكون نتيجته استدعاء وظيفة غير ضروري في كليبر 5.x ، إذ أن هذا الأمر ستتم معالجته في وظيفتين من وظائف كليبر هما: DEVPOS( ) (والتي تضع المؤشر في مكانه) ، والوظيفة DEVOUT( ) (والتي تعرض القيمة).

ونحن ليس بحاجة فعلياً لعرض أية قيمة في هذه الحالة ، ولذلك ، يمكننا حذف استدعاء وظيفة DEVOUT( ) ، وكل مايلزمنا فقط هو الوظيفة DEVPOS( ) لإعادة موقع المؤشر إلى ماكان عليه سابقاً.

(كما يرجى ملاحظة أن استخدام الوظيفة ( )SETPOS في كليبر سيكون أفضل أيضاً للتمثيل على هذه الحالة ، إذ أن الوظيفة ( )SETPOS هي متخصص بالشاشة فقط بغض النظر عن تجهيزات الوسائل الأخرى).

```
oldrow  := row( )
oldcol   :=  col( )
*
setpods( oldrow , oldcol )
```

أما إذا كنت حريصاً على وضوح القراءة (ولانجد أي داعٍ لذلك) فيمكنك استخدام المعالج الأولي من خلال كتابة أمر من قبل المستخدم يسمى: MOVE CURSOR على النحو التالي:

```
#xcommand MOVE CURSOR TO <r> , <c> = > setpos<r> , <c> )
oldrow  := row( )
oldcol   :=  col( )
*
move cursor to oldrow , oldcol
```

كما أنه يمكننا أعطاء مثال أخر على رفع مستوى فعالية البرنامج إلى أقصى حد من خلال الأمر CLEAR SCREEN ، فقد درج المبرمجون على استخدام هذا الأمر بدلاً من استخدام الأمر CLEAR في الإصدارات السابقة من كليبر. في حين أن كليبر 5.X قد بسَّط الأمر كثيراً باستخدام CLS والذي يترجمها المعالج الأولي على أنها العبارة الكاملة CLEAR SCREEN.

إلا أنك قد تفهم أن أمر CLS يترجم إلى استدعاء وظيفتين هما:

```
#command  CLS                    ;
          =>  Scroll( )          ; ;
              SetPos( 0,0 )
```

كما أن الوظيفة ( )SCROLL في كليبر 5.x، دون أية متغيرات تمسح محتويات الشاشة جميعها. أما الوظيفة ( )SETPOS فإنها تضع المؤشر في أعلى يسار الشاشة. ويمكن أن نعدّ على الأصابع فقط المرات التي يمكن أن يحتاج المبرمج فيها إلى تغيير موضع المؤشر بعد مسح محتويات الشاشة. لذا ، فإن هذه الوظيفة ( )SETPOS ليست ضرورية على

الإطلاق. وقد ترى الاتجاه لاستخدام الوظيفة ( )SCROLL بدلاً من استخدام أمر CLS كلما دعتك الضرورة لذلك.

ملاحظة لمستخدمي كليبر 5.2

تقبل الوظيفة ( )SCROLL في هذا الإصدار متغيراً سادساً اختيارياً وهو <nColumns> ، فإذا تم تعريفه فستدور الشاشة أفقياً scroll (عرضيا) بعدد الأعمدة الذي تم تحديدها. وتسبب الأرقام الموجبة تدوير الشاشة إلى اليسار ، بينما تسبب الأرقام السالبة تدويرها إلى اليمين. ويبين المثال التالي أدناه عمل هذه الوظيفة.

```
function main
local x
@ 0 , 75 , 24 , 79 box " ********* "
for x := 1 to 15
    scroll( , , , , , 5 )
    inkey ( 01 )
next
return nil
```

٤) توسيع لغة كليبر

إذا كنت تعرف الوظائف التي تنفذها باستخدام أوامر محددة فيمكنك كتابة موجهاتك البديلة للمعالج الأولي والذي يمكنه أن يبني على تلك الأوامر. ولعل أفضل مثال على هذا هو أمر INDEX ON :

```
index on Keyfield to indexfile
```

وسيترجم هذا من قبل المعالج الأولي على النحو التالي:

```
dbCreateIndex (  "indexfile",  "Keyfield",  { | |  Keyfield  }, ;
                 if( .F. , .T. , NIL ) )
```

ولقد تمت إضافة وظيفة dbCreateIndex( ) في كليبر 5.2 وتم توثيقه في دليل نورتون (Norton Guides). أما المتغير الذي نهتم به هنا فهو كتلة الشيفرة. وسيتم تقييم هذا لكل سجل من سجلات قاعدة البيانات لإنشاء ملف الفهرسة.

أن كل ما تفعله كتلة الشيفرة باستخدام الموجه INDEX TO هو أن تعيد القيمة لمفتاح الحقل. ومن السهولة بمكان للمبرمج أن يدخل استدعاء وظيفة أمام تعبير ذاك المفتاح key expression ، حيث تقوم تلك الوظيفة عندئذ بعرض "سطر الحالة" status bar. فإذا تم الأنتهاء من عملية الفهرسة فلن تكون هذا الوظيفة متعلقة بأية طريقة من الطرق بملف الفهرسة.

```
{ | | indexbar( ) , Keyfield }
```

تعرض وظيفة IndexBar( ) "سطر الحالة" الشهير والذي يبين للمستخدم التقدم النسبي لعملية الفهرسة. ويكون لهذا العرض وقع جيد على المستخدم الذي يلاحظ بعينيه مدى التقدم في عملية الفهرسة بشل ملموس ويدرك أن الكمبيوتر يقوم بالعمل المطلوب منه.

ولايعني هذا أنك ستكتب كتل شيفرة وتجعل برامجك أكثر تعقيداً مما هي عليه. بل قد يتساءل متسائل: لماذا نحتاج أن نفعل مثل هذا طالما أن المعالج الأولي سيقوم بتنفيذ هذه الأعمال بدلاً عنا ؟ ولكن بدلاً من ذلك ، يمكن أن تكتب أمراً يعده المستخدم على شكل موجه معالج أولي على النحو التالي:

```
#xcommand INDEX ON <key> TO <file> GRAPH [ <u :  UNIQUE> ] ;
        => dbCreateIndex( < ( file ) > , < "key " > ,  ;
                                   { | | indexbar( ) , < key > } , <.u.> )
```

وإن الأختلافات بين هذا الأمر والأمر الموجود في كليبر هي مايلي: (أ) الكلمة المفتاح GRAPH ، (ب) وظيفة شريط الفهرسة IndexBar( ) الذي تتضمنه كتلة الشيفرة. وسنناقش وظيفة شريط الفهرسة IndexBar( ) أثناء مناقشة كتل الشيفرة وستجد هذه الوظيفة على الأسطوانة المرفقة بالكتاب.

تجاوز حد الذاكرة Memory Overbooked

عندما تبدأ باستخدام ملفات التضمين #include فقد تواجهك رسالة خطأ وقت التجميع مزعجة. ويحدث هذا لأن كليبر يسمح لك بالتعامل مع ٦٤ كيلوبايت من موجهات المعالج الأولي في أي وقت من الأوقات. ويحدث هذا عادة لدى استخدام ملف ترويسة ضخم جداً. وإن أحسن حل لهذه المشكلة هو أن تقسم ملف الترويسة الضخم هذا إلى عدة أجزاء تطابق الأقسام المختلفة من برنامجك. ثم ضمّن الأجزاء اللازمة فقط في كل قسم من أقسام البرنامج.

تحذير الشيفرات الميتة Dead Code Caveat

إن كليبر 5.x ، كما أشرنا آنفاً ، هو ذكي نسبياً بحيث يمكنه إزالة أية شيفرة من ملف OBJ والتي لا يمكن تنفيذها بحال من الأحوال أثناء وقت تشغيل البرنامج مثل الشيفرة التالية:

```
if .f.
  ? "I may walk , but I will never run"
endif
```

ويعتبر هذا الأمر جيداً إلى حد ما في معظم الأحيان إذ يصبح كل من ملفي الأهداف والتنفيذ متشابهين ، إلا أن هذا قد يسبب مشاكل عندما تبدأ بإعداد موجهات المعالج الأولي الخاصة بك.

ويبين المثال التالي الأمر السابق. حيث أن الأمر القياسي في كليبر SET INDEX TO يقوم دائما بمسح الفهارس النشطة حالياً باستخدام الوظيفة dbClearIndex(). وسنعد نسخة من ذاك الموجه ونحاول إضافة الفقرة ADDITIVE:

```
#command SET INDEX TO [ <(index1)> [ , <(indexn)> ] ] [ <add :ADDITIVE> ] ;
        => IF( ! <.add.> , dbClearIndex( ) , NIL )
```

```
    [ ; dbsetIndex ( < ( index1 ) >  ) ]
    [ ; dbsetIndex ( < ( indexn ) >  ) ]

function main
set index to ndx1 , ndx2  additive
return nil
```

وقد يبدو هذا العمل واضحاً ومباشراً ، إلا أنك إذا حاولت تجميع هذا البرنامج فستحصل على الخطأ التالي:

Error C2003 Untrapped syntax error in statement

وإذا نظرنا بمزيد من التفصيل إلى هذا فسنجد أننا إذا لم نحدد الفقرة ADDITIVE فإن معلّم النتيجة المنطقي logify result marker سيتسبب بإنتاج الشيفرة التالية:

```
if ( !  .f. , dbClearIndex( ) , NIL ) . . .
```

ولذلك ، سيستدعى أمر ( )dbClearIndex لمسح الفهارس ، إلا أنك إذا استخدمت الفقرة ADDITIVE سيتم إنتاج الشيفرة التالية أيضاً:

```
if ( !  .f. , dbClearIndex( ) , NIL ) . . .
```

ويعني هذا بوضوح أن الأمر السابق ( )dbClearIndex لن يُستدعى ، إلا أن الأقل وضوحاً من هذا (والأكثر أهمية منه في الوقت ذاته) أن هذا سيصبح في عِداد "الشيفرة الميتة" Dead Code. وبهذا فإن المجمِّع لن يزيل وظيفة ( )dbClearIndex فقط بل سيزيل كل ماهو مثبت فيها ضمنياً أيضاً أي العبارة الشرطية ()IF بالاضافة إلى (!). والذي يبقى مايلي فقط:

```
NIL
```

وهذا ماسيسبب خطأ المجمِّع ، والأمر المحيِّر في هذا السيناريو هو أنه لايتم إظهار أي من أعمال تحسين الأداء هذه ، في ناتج المعالج الأولي ، وبهذا فإن اختبار ملف PPO. لن يظهر أي أمر غير عادي قد وقع فعلاً. لذلك نقترح أثناء الحصول على مثل رسالة الخطأ

هذه من المجمّع في سطر يتعلق بموجه المعالج الأولي نقترح أن تنظر بدقة لتبحث عن إمكانية وجود "شيفرة ميتة".

أمثلة عن المعالج الأولي Preprocessor Examples

كتابة برامج ثنائية اللغة

لنفترض أنك تعدُّ برنامج تسويق سيستخدم من قبل أشخاص ناطقين بالأنجليزية أو بلغة أخرى. فهناك ثلاثة طرق يمكنك استخدام أي منها لإعداد مثل هذا البرنامج:

١- إعداد نسختين مستقلتين تماماً من هذا البرنامج. ولن يكون هذا الحل عملياً تماماً بل سيكون غير ذي قيمة على الإطلاق تقريباً.

٢- احذف كل النصوص الساكنة (واجهة المستخدم user interface) من البرنامج ثم أنشيء نصاً باستخدام ملفات DBF. أو ملفات MEM. لكل لغة تريدها. ثم أعد تركيب البرنامج من جديد بحيث يقرأ "المتغيِّرات" من هذه الملفات في البداية ، ثم يستخدمها خلال تنفيذ البرنامج جميعه.

ولاشك أن هذه الطريقة تفوق إلى مالانهاية الطريقة الأولى ، إلا أن هناك احتمالات لعدد من المحاذير ، والتي تتضمن ضريبة في مستوى الأداء التي سوف تدفعها لاستخدام المعلومات الموجودة على القرص الصلب ، وكذلك ضريبة أخرى وهي مشاكل استخدام ملف واحد من قبل عدة أشخاص في الوقت ذاته على شبكة اتصالات محلية. إلا أن هذا الحل معقول تماماً لمستخدم واحد على جهاز ذي وحدة معالجة معقولة مثل ٣٨٦ أو أعلى من ذلك.

٣- استخدام برنامج ماقبل المعالجة

أما الخيار الثالث ، فلا بد من نقاشه بشكل موضوعي تفصيلي ، وهذا ما أردناه من هذا البحث هنا ، وهو أمر سهل جداً ، إذ أننا سنعتمد على الموجه #ifdef والذي تحدثنا بشكل موجز عن إمكاناته سابقاً. فعلى سبيل المثال:

```
#ifdef ARABIC

#define M_NETERR        " لايمكن إقفال السجل في هذا الوقت "
#define M_CONTINUE      "هل تريد الإستمـرار ؟ (نعم/لا) "
#define M_TOF           " بدايـة الملف! "
#define M_BOF           " نهاية الملف!  "
#define M_PRINT         " هل تريد الطباعة إلى الطابعة أو إلى الملف ؟ "
#define M_CONFIRM       " هل أنت متأكـد؟ "
#define M_NOTFOUND      " غيـر موجودا "
#define M_ADDING        "إضافة سجل-اضغط مفتاح  للحفظ أو مفتاح  للخروج"
#define M_EDITING       "تعديل سجل-اضغط مفتاح  للحفظ أومفتاح  للهروب"

#else

#define M_NETERR        " Could not lok record at this time "
#define M_CONTINUE      " Would you like to continuer ?  (Y/N ) "
#define M_TOF           " Top of file ! "
#define M_BOF           " Bottom of file !  "
#define M_PRINT         " Print to printer or file ? "
#define M_CONFIRM       " Are you sure ? "
#define M_NOTFOUND      " Not Found ! "
#define M_ADDING        " Add record - ^w to save;   ESC to exit "
#define M_EDITING       " Edit record - ^w to save:  ESC to exit "

#endif
```

ثم تشير إلى هذه الرسائل باستخدام ثوابت بيانيـة manifest constants تقوم بتعريفها.
مثلاً إليك الشيفرة المستخدمة لقائمة الاختيارات التي علىشكل شريط مضاء:

```
@  23, 0            prompt  M_ADD
@  23, col ( ) + 2  prompt  M_EDIT
@  23, col ( ) + 2  prompt  M_SEARCH
@  23, col ( ) + 2  prompt  M_DELETE
@  23, col ( ) + 2  prompt  M_NEXT
@  23, col ( ) + 2  prompt  M_PREV
```

```
@  23, col ( ) + 2      prompt  M_QUIT
menu  to  key
```

ويمكنك الآن الانتقال من لغة إلى أخرى بكل بساطة ودون أي جهد باستخدام المفتاح /D من خيارات المجمّع. فإذا أردت استخدام اللغة العربية ، جمّع باستخدام الأمر التالي :

```
clipper  myprog  / dARABIC
```

وسيكتشف المعالج الأولي وجود معرف اللغة العربية ARABIC . أما إذا أردت التجميع بالأنجليزية ، فلا تستخدم الخيار السابق وسيستخدم البرنامج اللغة الأساسية وهي الأنجليزية في هذه الحالة.

مولد التقارير جرمبفيش Grumpfish Reporter

لقد استخدم المعالج الأولي أثناء إعداد وتطوير " مولد التقارير جرمبفيش" وخاصة فيما يتعلق بالتجميع الشرطي. ويعتبر هذا البرنامج " مولد التقاريرجرمبفيش " برنامجاً مرناً ، قابلاً للربط ، رائعا لكتابة الاستعلامات والتقارير ، والذي تمت كتابته بشكل كامل باستخدام كليبر 5.x. ومع أن الحديث عن هذا البرنامج ومزاياه سيستغرق كثيراً من الوقت ، لذا سنكتفي بإيراد أهمها فقط فيما يلي:

(أ) احتواؤه على المساعدة الفورية المتعلقة بالموضوع مباشرة. (ب) إمكانية إخراج صفحات البيانات الإلكترونية وملفاتها(بواسطة المنتج الآخر CLIPWKS®. (ج) احتواؤه على غلاف دوس Dosshell . (د) دعم مذكرات برنامج Flexfile™.

ومع أن هذه المزايا جميلة جداً ، فلا نجد أن كل مستخدم لهذا البرنامج يحتاج إليها -أو يستخدمها. وبما أن شيفرة المصدر موجودة بالكامل مع المنتَج ، فسيكون من واجب المطوِّرين الاطلاع على المزايا جميعها والتعرف على مايلزمهم وما لايلزمهم من المزايا التي يحتوي عليها البرنامج. ومع ذلك ، فإن المعالج الأولي يجعل هذه العملية وكأنها ليست هناك ، أي في منتهي السهولة ، إذ يمكّن المطوِّر من لفّ أقسام البرنامج المتعلقة بكل ميزة من تلك المزايا باستخدام عبارتي: #ifdef و #enif . ويمكن هذان الأمران المطور من

تبسيط العمليـة بوضـع العبـارات الأربعـة التاليـة في ملـف ترويسـة تقريـر أساسـية وهـي) على النحو التالي: GR.CH (

```
#define     USING_HELP
#define     SPREADSHEET_OUTPUT
#define     DOS_SHELL
#define     FLEXFILE_SUPPORT
```

وكل مايجب على مطوِّر البرنامج أن يفعله هو أن يعلّق على العبارات المطابقة للمزايـا الـتي لا يريد استخدامها ، ثم يجمِّع البرنامج ، حيث يقوم المعالج الأولي بحذف ما لا حاجـة لـه. وسيوفر هذا العمل كثيراً من الوقت والجهد على العاملين في هذا الموضوع جميعهم.

التعليق على استدعاءات الوظائف الفردية

لنفترض أن لديك جزءاً من شيفرة المصدر تستدعي الوظيفـة ( )SomeFunc باسـتمرار. ولنفترض أيضاً أنـك ، ولسـبب مـن الأسـباب ، تريـد أن تعلّـق عمـل هـذه الاستدعاءات الوظيفية لفترة من الفترات. فيمكنك استخدام البحث الشامل واستبداله لتعليق عمل كل سطر من تلك السطور. إلا أن الطريقة الأسهل بكثير لتنفيذ مثل هذا العمـل هـي أن تضـع العبارة التالية في أول ملف PRG. :

```
#xtranslate  SomeFunc( [ <params , . . . > ] )  =>
```

أولاً: إذا استخدمت قائمة معلم المطابقة list match-marker فلـن تهتـم بعـدد المتغـيرات التي تحدد لهذه الوظيفة. ثانياً: إذا وضعت معلـم المطابقـة ضمـن قوسـين فـإن المعـالج الأولي سيعتبره "اختياريـاً" ، أي : سـيحذف اسـتدعاءات ( )SomeFune حـتى لـو لم تضـع أيـة متغيرات.

ونقترح استخدام هذه الطريقة العملية إذ أنها ستوفر عليك الكثير من الوقت وستسـاعدك في تنفيذ أعمالك على الوجه المطلوب أيضاً. فقـد يكـون لديـك علـى سـبيل المثـال وظيفـة لكتابة المدخلات إلى ملف نص text file في مواقع مختلفة من البرنامج ، على النحو التالي:

```
#ifdef  DEBUG
   profiler(procname( ) ,  procline( ) ,  variable_name ,  more_info . . . )
#endif
```

فيمكنك في هذه الحالة إبقاء هذه الشيفرة على حالها تماماً بأن تسبقها بعبارة: #ifndef DEBUG... #endif ، إلا أن هذه العبارة ستكون أكثر تحديداً وسهولة إذا وضعت في أول كل ملف برنامج PRG. يتعلق بها ، العبارة التالية:

```
#ifndef  DEBUG
   #translate   profiler(  [   < stuff , . . . >   ]  )  =>
#endif
```

ويمكنك الأنتقال جيئة وذهاباً مابين الوضعية العادية ، ووضعية "التلخيص profile" بتجميع ملفات برامجك باستخدام الخيار dDEMO/ .

تعليقات الكتلة المتداخلة Nested Block Comments

لعلك تعرف أن بمقدورك إيقاف عمل كتلة شيفرة ما باستخدام خيار "*/" وخيار "/*" فأما الخيار الأول فيشير إلى بداية تعليق الكتلة ، بينما يحدد الخيار الثاني نهايته. مثال:

```
// هذه هي الطريقة المملة لتعليق
// عمل أكثر من سطر من سطور الشيفرة

/*
  وهذه الطريقة هي الأكثر سهولة من الطريقة السابقة
  لتعليق عمل أكثر من سطر من سطور الشيفرة مؤقتاً
*/
```

ولعل هذه هي أسهل طريقة للتعليق على أكثر من سطر من سطور البرنامج. إلا أن هناك أثراً سلبياً واحداً لاستخدام الخيار "*/" و "/*" ، فهي لا يمكن تداخلها. إلا أن هناك

طريقة بسيطة يمكن استخدامها لتجاوز هذه العقبة وهي استخدام المعالج الأولي لكليبر 5.x.

فلنفترض أن لديك كمية كبير من شيفرة ما تريد إيقاف عملها مؤقتاً وقد يحتوي هذا الجزء من الشيفرة على تعليق لكتلة واحدة ، أو أكثر داخله. ضع موجهاً من نوع #ifdef قبل الجزء المراد من الشيفرة مستخدماً اسماً لثابت بيان لايحتمل وجوده إطلاقاً. ثم ضع الموجه الآخر المكمل #endif في نهاية ذلك الجزء من الشيفرة التي تريد إيقاف عملها مؤقتاً.

```
#indef  BLAHBLAHBLAH
    //  أوامر
    /*
        تعليق
    */
    //  المزيد من الأوامر
#endif
```

ولعل هذه الطريقة أسرع بكثير من استخدام "تعليقات سطرية" على كل عبارة تريدها أن تخضع للذلك الموجه للتوقف.

اختبار المتغيرات باستخدام NIL

لقد كان يتم اختبار المتغيرات التي تمرر للوظائف في الإصدارات السابقة من كليبر باستخدام الوظيفة PCOUNT( ) والوظيفة TYPE( ) بصورة رئيسة. ومع أننا تعلمنا واعتدنا التعامل مع هذه الترتيبات ، فقد لاحظنا أن هذين الوظيفتين أصبحتا غير مستعملتين مع نوع بيانات NIL.

ويتجاوز كليبر ، الإصدار الجديد ، المتغيرات غير الضرورية بوضع فاصلة في قائمة المتغيرات. ولن تحتاج لاستعمال سلسلة صفرية null string كما كان الحال في كليبر

Summer'87. فإذا حذف متغير في القائمة الأساسية فسيتم تأسيسه داخل الوظيفة بقيمة "صفر" null فقط.

تبين هذه الوظيفية ، والتي تعرض مربعاً ضمنه رسالة على الشاشة ، أنها قبلت خمسة متغيرات. وسنفترض أن المتغير الأول (وهو الرسالة) سيتم إرساله بشكل دائم. أما المتغيرات الأخرى فهي اختيارية. ويؤثر المتغيران الثاني والثالث على موقع كل من السطر والعمود في مربع الرسالة. فإذا لم يتم إرسال هذين المتغيرين فسيكون مربع الرسالة في منتصف السطر. وأما المتغيران الثالث والرابع فسيؤثران على لون المربع ، وعلى لون الرسالة. فإذا لم يتم إرسال هذين المتغيرين. فسيتم تعيين قيمة الصفر لهما في وظيفة ShowMsg( ). وبهذا يصبح أمر اختبار وصول إرسال متغيرات الرسالة أمراً سهلاً جداً.

```
function showmsg( msg , boxcolor , msgcolor )
local oldcolor , buffer
if boxcolor == NIL  // assign box color if not passed
   boxcolor  := 'w/r'
endif
if msgcolor == NIL  // assign message color if not passed
   msgcolor := 'w/r'
endif
// etcetera
```

ويمكن جعل هذا الأمر أكثر تحديداً أيضاً باستخدام المعالج الأولي. فبدلاً من استخدام أمر IF..ENDIF لكل متغير. يمكننا بسهولة تعريف أمر معرف من قبل المستخدم يقوم بهذه العملية بدلاً عنا ، وهو على سبيل المثال ، على النحو التالي:

```
#xcommand  DEFAULT <p> TO <v>  [, <p2> TO <v2> ] =>  ;
                IF <p> == NIL ; <p> := <v>   ;          END ;
                [; IF <p2> == NIL ; <p2> := <v2> ;    END ]
```

حيث يقوم هذا الأمر باختبار المتغير المسمى (<p>) مقابل NIL ، ويعين له قيمة مفترضة هي (<v>) إذا كانت هي القيمة المطلوبة ، وإلا فستبقى القيمة كما هي دون تغيير.

إن استخدام الأقواس في النص الناتج يشير إلى أنه يمكن إعادة الفقرة الاختيارية وهكذا يمكنك وضع سلسلة من القيم معاً في العبارة ذاتها ، كما سنرى في الشيفرة التالي. وهناك ملاحظتان إضافيتان عن هذا التركيب:

١) يحتمل أنك رأيت أمثلة أخرى للموجه DEFAULT TO باستخدام التعيين السطري المباشر مع العبارة الشرطية IF( ) المضمنه داخلياً. ومع ذلك ، فقد تزيد السرعة مابين ٥-١٠٪ للقيام بهذا التعين عند الضرورة فقط ، ولهذا السبب بالذات فإننا نقدم عبارة IF..ENDIF . أما في كليبر 5.2 : فإن الموجه DEFAULT TO قد أصبح جزءاً من ملف الترويسة COMMON.CH.

٢) لماذا نستخدم END بدلاً من ENDIF ؟ فالإجابة تكمن في الموجه ENDIF ذاته كما هو معرف في ملف الترويسة STD.CH :

```
#command   ENDIF        < * x * >        =>  endif
```

وكما أشرنا آنفاً ، فإن معلم المطابقة العشوائي wild match-marker يعني أن المعالج الأولي سيحذف أي شيء يتبع يعقب ENDIF بما في ذلك (وللأسف) نص المخرجات الاختيار المتكرر.

التنسيق الحر لقائمة المتغيرات

إلحاقاً للمثال الذي قدمناه آنفاً عن الوظيفة ShowMsg( ) ، يمكنك الاستغناء عن الحاجة لتذكُر ترتيب المتغيرات بتركيب موجه المعالج الأولي #xcmmand بحيث يشبه هذا الموجه مايلي:

```
#xcommand        MESSAGE  <msg>
                [ ROW   <row>    ]
                [ COL   <COL>    ]
                [ BOXCOLOR  <boxcolor>  ]
                [ MSGCOLOR <msgcolor> ]
                [ <double: DOUBLE>          ]

      msgbox( <row>,  <col>,  <msg> ,  <boxcolor> ,
                 <msgcolor> ,  <.double.>  )
```

ويجب أن تحاط كـل فقـرة بـأقواس بحيـث تشـير إلى أنهـا اختياريـة ، وتصبـح الآن حـراً لاستخدام هذه الفقرات بأي ترتيب تريده. فمثلاً ، تصبح كـل الأمـور التاليـة صحيحـة ومقبولة:

```
message   "Hello there" // use default box and message colors
message   "Hello there" msgcolor 'w/r'  // use default box colors
message   "Hello there" msgcolor 'w/r' boxcolor 'w/b'
```

ولعلك تريد بعد كتابة الوظيفة ShowMsg( ) اختبار كل متغير من هذه المتغـيرات لقيمـة NIL ، ثم تعيين القيم المفترضة إذا لزم الأمر. ولعل هذا في رأينـا أفضـل أسـباب اسـتخدام الموجه #xcommand بدلاً من الموجه #xtranslate .

```
#include  "box.ch"

#xcommand       DEFAULT <p> TO <v> [ , <p2> TO <v2> ] =>
                IF <p> == NIL ; <p> := <v> ; END ;
                [ ; IF <p2> == NIL ; <p2> := <v2>  ; END ]

#xcommand MESSAGE <msg>
                [ ROW <row> ]
                [  COL  <col>  ]
            [  BOXCOLOR  <boxcolor> ]
            [  MSGCOLOR  <msgcolor> ]
            [  <double : DOUBLE>  ]
                                                          =>
   msgbox( <row> , <col> , <msg> , <boxcolor> , ;
       <msgcolor> , <.doubie.> )

function main
message  "Hello" row 10 double
message  "Hello there" col 20 msgcolor 'tw/r'
message  "Hello there" boxcolor '+w/b' row 10
return nil

function msgbox ( nRow , nCol , cMsg , cBoxColor , cTextcolor , lDouble )
default nRow to maxrow ( ) / 2 - 1
default  nCol to int ( maxcol ( ) - len ( cMsg ) / 2 )
default  cBoxColor to 'w/b'
default  msgcolor to 'w/b'
@ nRow , nCol , nRow + 2 , nCol + len( cMsg ) + 2  ;
            box if ( 1Double , B _ DOUBLE , B _ SINGLE ) + '  '
            color cBoxColor
@ nRow  + 1 , nCol  + 1 say cMsg color cTextcolor
return nil
```

رسم المربعات Box Drawing

إذا كنت قد تعبت من رسم المربعات ضمن إطارات كما حصل لي ، فإليك بعض الأوامر المعرفة من قبل المستخدم وثوابت البيان لمعالجة هذا الموضوع.

وقبل المضيّ في هذا الموضوع ، يجب أن تتذكر أن كليبر يتضمن ملف ترويسة اسمه BOX.CH يحتوي على عدة ثوابت بيان مفيدة تمثل إطارات لمربعات. إلا أن شكوانا الرئيسية من تعاريفهم أن أياً من هذه التعريفات لا يحتوي على الرمز التاسع (الذي يملأ المربع) لذلك فإما أن تضيفه أنت ، أو يكون الجزء الداخلي من المربع فارغاً دون لون. لذا، فإننا نقترح أن تصمم ملف ترويسة مربع خاص بك لتخفف أعباء البرمجة عنك ، على النحو التالي:

```
#define  B_DOUBLE  chr(201) + chr(205) + chr(187) + chr(186) +  ;
           chr(188) + chr(205) + chr(200) + chr(186) + chr(32)

#define  B_SINGLE   chr(218) + chr(196) + chr(191) + chr(179) +  ;
           chr(217) + chr(196) + chr(192) + chr(179) + chr(32)

#define  B_DOUBLESINGLE  chr(213) + chr(205) + chr(184) + chr(179) +  ;
           chr(190) + chr(205) + chr(212) + chr(179) + chr(32)

# define  B_SINGLEDOUBLE chr(214) + chr(196) + chr(183) + chr(186) +  ;
           chr(189) + chr(196) + chr(211) + chr(186) + chr(32)

#define  B_THICK  chr(219) + chr(223) + chr(219) + chr(219) +  ;
           chr(219) + chr(220) + chr(219) + chr(219) + chr(32)

#define  B_NONE    space( 9 )

#xtranslate  SingleBox( <top> ) , <left> , <bottom> , <ribht>  [ ,  <color> ] )  =>  ;
              DispBox(<top> , <left> , <bottom> ,  <right>  B_SINGLE ,  <color> )

#xtranslate DoubleBox(<top>  , <left> , <bottom> ,  <right>  [ , <color> ] )  =>  ;
              DispBox( <top> , <left> , <bottom> ,  <right> , B_DOUBLE , <color>)

function  test
SingleBox( 0 ,  0 ,  maxrow( ) ,  maxcol( ) ,  ' w / r '  )
DoubleBox( 5 , 19 , 12 , maxcol( )  - 10 ,       ' w / r '  )
@  11,   34,  13,  45  box  B_THICK
@  12,   36, say  "Hi  there"
```

```
inkey(0)
return nil
```

إضافات امتدادات لأسماء الملفات

تطلب معظم برامج كليبر من المبرمج إدخال نهاية لاسم الملف. وغالباً ما تسمح بإدخال امتداد اختياري. إلا أنها إذا لم تفعل ذلك. فإن البرنامج ذاته سيضيف نهاية لاسم الملف الذي تعده. ويقوم الأمر التالي بهذا العمل لأجلك.

ملف المصدر الأصلي (.PRG)

```
#translate  AddExtension( < file >,  <ext> ) =>  ;
              < file >  :=  upper(  < file >   )  +  if( !  " . "  + upper(  < file >  )  $  ;
                                      upper(  < file >   ) ,  " . "  + upper(  < file > ) ,  ' ' )

AddExtension( mfile,  ' dbf ' )
```

ملف مخرجات المعاج الأولي (.PPO)

```
mfile  :=  upper(mfile)  + if  (  !  "."  + upper( "dbf" )  $  upper(mfille)  ,   ;
             "."  + upper( "dbf" ) ,  " " )
```

لم تعد الوظيفة STRPAD( ) موجودة

إذا استخدمت هذا الأمر في إصدار كليبر Summer'87 فلعلك تكون قد اكتشفت أنه غير موجود في كليبر 5.x ، إلا أن كليبر يقدم وظيفة بديلة لها وهي PADR( ) والتي تفعل كل شيء يمكن أن تفعله الوظيفة السابقة STRPAD( ). وبدلاً من أن تغير كل موقع في شيفرة المصدر تظهر فيه الوظيفة القديمة STRPAD( ) إلى الوظيفة الجديدة PADR( ) فيمكنك كتابة ترجمة بسيطة تقوم بذلك بدلاً منك على النحو التالي:

```
#xtranslate  strpad( <msg>,  <length> ) => padr( <msg>, <length> )
```

تعبيرات عامل البديل Alias

إن عامل alias وهو ("->") يمكّنك من الإشارة إلى حقل ما ، أو تقييم تعبير ما في منطقة عمل غير محددة. وسيتمكن هذا العامل من اختيار منطقة العمل المرغوبة ، والقيام بالعملية، ثم يعيد اختيار منطقة العمل السابقة. ويمكنك هذا من تجميع وضغط شفرتك بحيث لا تحتاج إلا عدة عبارات SELECT ظاهرة.

وتنطبق عبارة ALIAS على أمر SKIP فقط. ومع ذلك ، فإن الشيفرة التالية تضيفها إلى الوظائف المختلفة الأخرى التي تتعامل مع قاعدة البيانات. وإن الفكرة الأساسية هي إضافة الفقرة الاختيارية " [ALIAS<A>] " إلى نص الإدخال ، والفقرة الاختيارية الأخرى المطابقة "[<a>->]" للنص الناتج. ويجب كذلك أن تتأكد من إضافة استدعاءات الوظائف المعنية داخل الأقواس الكبيرة [ ]. على النحو التالي:

```
#xcommand  SEEK  <xpr>      [ALIAS  <a> ] => [<a> ->]  (dbseek( <xpr> ) )
#xcommand  GOTO <n>         [ALIAS  <a> ] => [<a> ->]  (dbGoto( <n> ) )
#xcommand  GO     <n>       [ALIAS  <a> ] => [<a> ->]  (dbGoto( <n> ) )
#xcommand  GOTO TOP         [ALIAS  <a> ] => [<a> ->]  (dbGOTOP( ) )
#xcommand  GO  TOP          [ALIAS  <a> ] => [<a> ->]  (dbGOTOP( ) )
#xcommand  GOTO  BOTTOM     [ALIAS  <a> ] => [<a> ->]  (dbGOBOTTOM( ))
#xcommand  GO  BOTTOM       [ALIAS  <a> ] => [<a> ->]  (dbGOBOTTOM( ) )
#xcommand  CONTINUE         [ALIAS  <a> ] => [<a> ->]  (dbContinue( ) )
#xcommand  APPEND BLANK     [ALIAS  <a> ] => [<a> ->]  (dbAppend( ) )
#xcommand  UNLOCK           [ALIAS <a>  ] => [<a> ->]  (dbUnlock( ) )
#xcommand  PACK             [ALIAS <a>  ] => [<a> ->]  (__dbPack( ) )
#xcommand  ZPA              [ALIAS <a>  ] => [<a> ->]  (__dbZap( ) )
#xcommand  DELETE           [ALIAS <a>  ] => [<a> ->]  (dbDelete( ) )
#xcommand  RECALL   [ALIAS <a>  ] => [<a> ->]  (dbRecall( ) )

function main
use invoices new
set index to invoices
use customer new
seek customer->custno alias invoices
delete alias invoices
go top alias invoices
return nil
```

وكما أشرنا سابقاً يجب ألا تعدّل ملف الترويسة المسمى STD.CH مباشرة. وإذا أردت استخدام عبارة ALIAS كما بينا هنا ، فيجب أن تعمل نسخة من ملف الترويسة STD.CH ، وسمّها باسم تعرفه ، مثلاً : ALIAS.CH ، ثم عدّل أوامر قاعدة البيانات طبقاً لذلك. وبعدئذ ، احذف كل شيء عدا الذي تم تغييره. وباستخدام أمر التضمين #include يمكنك تضمين ملف الترويسة هذا في برنامجك وبذلك ، ستلغي أوامرك الجديدة الأوامر المفترضة التي يفترضها كليبر.

استدعاء مزايا النص/اللون

إذا أردت فحص كل من مزايا النص / اللون في موقع ما على الشاشة يمكنك استخدام الوظيفة SAVESCREEN( ) لحفظ ذلك العنصر. ومع ذلك ، فبدلاً من حشد برنامجك بكثير من هذه الاستدعاءات ، استخدم الأوامر المعرفة من قبل المستخدم وهما TextAt( ) و ColorAt( ) ، على النحو التالي :

```
#xtranslate  TextAt( <r> , <c> )  =>   ;
               substr(savescreen( <r> , <c> , <r>  , <c> ,  1,  1 )

#xtranslate  ColorAt( <r> , <c> )  =>  ;
               color_n2s (  substr ( savescreen ( r > , < c > , < r > , < c > ,  2,  1 )

function  main
@ 1,1  say  ' testing '  color   ' w / b '
?  TextAt( 1,  3 )
?  colorAt( 1,  0 )
?  colorAt( 1,  4 )
return  nil
/*
     color_n2s( ) : convert  color  number  ( 0 - 127 )  to  DBASE  color  string

*/
function   color_n2s(colorno)
static  foreground  :=  ' N  B  G  BG  R  BR  GR  w  N+  B+  G+  BG+R+ '  +  ;
                                 ' BR+GR+W+ '
static  background  :=  ' N  B  G  BGR  BRGRW  '
local  blinking
if  valtype(colorno)  =  " C "
   colorno  :=  bin2i(colorno)
endif
```

```
blinking := (colorno > 127 )
colorno := colorno % 128
return ( if(blinking , '*' , ' ' ) + ;
        trim(substr(foreground , (colorno % 16 ) * 3 + 1, 3 ) ) + '/' + ;
        trim(substr(background , int(colorno / 16 ) * 2 + 1, 2 ) ) )
```

رئيس الفرقة الموسيقية

حتى لو كنت قد استبعدت من ذهنك تماماً احتمال التغذية الراجعة الصوتية في برنامجك فقد تبقى لديك الرغبة بالتعرف على أساليب البرمجة والاطلاع عليها بشكل عام. وقد ضمنت في هذا الكتاب ثلاثة مواضيع موسيقية هي: Charge و NannyBoo و TheFifth(وهي ألحان افتتاحية سيمفونية بتهوفن الخامسة). ولدى استدعاء أي من هذه الأوامر سيحولها المعالج الأولي إلى مصفوفة من مصفوفات متداخلة (تحتوي على ذبذبة اللحن ومدته) تمرّ بعدها إلى وظيفة الألحان ( )Tunes. والتي ليست هي وظيفة بمعنى الكلمة ، فلذلك يقوم المعالج الأولي بترجمتها إلى وظيفة ( )AEVAL والتي تستدعي بدورها وظيفة اللحن ( )TONE لتعزف لحناً واحداً لكل عنصر من عناصر المصفوفة.

وبما أن الموجه xcommand# هو المستخدم فيجب أن تبدأ أسماء هذه الألحان العبارة بدلاً من أن تكون موجودة داخلها على النحو التالي:

```
#xcommand charge => tunes( { 523,2} , {698,2} , {880,2} , {1046,4} , ;
                        {880,2} , {1046, 8} } )
#xcommand NannyBoo => tunes( {196,4} , {196,4} , {164,4} , {220,4} , ;
                        {196,8} , {164,8} } )
#xcommand TheFifth => tunes( {392,2} , {392,2} , {392,2} , {311,10} , ;
                {15,12} , {349,2} , {349,2} ,{349,2} , {293,2} , {311,10} , ;

#translate tunes( <a> => aeval( <a> , { | a | tone(a\[1} , a\[2] ) } )

function music
charge
inkey ( 0 )
nannyboo
inkey ( 0 )
thefifth
inkey ( 0 )
return nil
```

ولاتقتصر هذه البرامج على إتحاف أسماع مستخدمي برامجك ببعض الألحان الموسيقية فحسب ، بل سترغبك باستخدام الوظيفة ( )AEVAL والمصفوفات المتدخلة أيضاً. ويرجى الانتباه إلى أن المثال المستخدم هو مجرد مثال فقط. وإذا كنت ترغب باستخدام مثل هذه الأمور مستقبلاً بشكل متكرر في برنامجك ، فيستحسن استخدامها كوظائف ، بدلاً من استخدامها كموجهات في المعالج الأولي. وبهذا تحد من عدد مرات ظهور هذه الشيفرات في ملفك التنفيذي.

نظام قوائم الاختيارات

تعتبر الوظيفة ( )LiteManu وظيفة قوائم ذات ثلاث التفافات جميلة هي:

- تضيء الحروف التي يطلب الضغط عليها للابتداء بتنفيذ الأمر.
- تزودك بحروف بدء بديلة.
- تستخدم القواعد اللغوية القياسية لكليبر.

قبل إصدار كليبر 5.x كان من الضروري تحميل كافة المصفوفات التي تحتوي على معلومات عن قوائم الاختيارات يدوياً . بل الأدهي من ذلك ، أن مصفوفات كليبر Summer'87 كانت ذات بُعدٍ واحد فقط مما حدّ فائدتها جداً.

وقد أفدنا من الفرصة السانحة لاستخدام المعالج الأولي لإعادة تعريف كل من الأمرين القياسيين: @..PROMPT , MENU TO لاستخدام LiteMenue. وإن استخدام هذا الأمر في المعالج الأولي شبيه جداً بالحالة التي يستخدم فيها أمرا: READ و @..Get بالإضافة إلى أن المصفوفات المتعددة الأبعاد وكتل الشيفرة جعلت المهمة أسهل بكثير.

وإليك أوامر كليبر القياسية @..PROMPT و MENU TO (كما أخذت في ملف الترويسة STD.CH):

```
#command @ <row> , <col> PROMPT <prompt> [MESSAGE <msg> ]  ;
            =>__AtPrompt( <row> , <col> , <prompt> , <msg> )
#command MENU TO <v>  ;
        => <v> := __MenuTo( { |_1 | if(PCount( ) == 0 , <v> , <v> :=_1) } , ;
                                          #< v >
```

ونبين فيما يلي إعادة تعريف هذه الأوامر على النحو التالي :

```
#command @ <row> , <col>  PROMPT <prompt>  [MESSAGE <msg> ]  ;
             [ ACTION < action > ]                          =>      ;
             IF(menulist == NIL,  menulist := { }, NIL );            ;
  AADD(menulist , { <row> ,  <col> , <prompt> ,  <msg> ,  < {action} > } )

#command MENU TO < v > =>  ;
             LiteMenu(menulist ,  @< v > ,  #< v >   ;  menulist  :=  { }
```

ومرة أخرى نذكّرك – عزيزي القارىء ونقترح أنه بدلاً من أن تجري التعديلات على ملف الترويسة STD.CH ، يمكن أن تضع هذه الأوامر المعدلة في ملف ترويسة خاص بك وتسميه MYMENU.CH وتحفظه مع بقية ملفات ترويساتك في كليبر 5.x.

وتكون القاعدة اللغوية للأمر الجديد على النحو التالي:

```
@  <r>,<c> PROMPT <prompt>  [MESSAGE  <message> ]  [ACTION  <action> ]
```

حيث يمثل كل من < r > ، < c > تعابير رقمية تبدأ كلاً من السطر والعمود اللذين يجب عرض عنصر قائمة الاختيارات عليهما.

أما <prompt> فهو تعبير حرفي يمثل اسم الاختيار ذاته. ولتحديد حرف بدء بديل اسبق الحرف المطلوب بالمدة (" ~ ") (المفتاح الموجود في أعلى يسار لوحة المفاتيح) في أمر <prompt>.

العبارة الاختيارية <message> هي تعبير حرفي يمثل الرسالة التي ستعرض عند إضاءة خيار قائمة الاختيارات المطابقة.

وأما العبارة <action> فهي اسم لوظيفة سيتم استدعاؤها لدى اختيار خيار مطابق من قائمة الاختيارات. ولا داعي لوضع هذه ضمن علامات تنصيص " " لأن المعالج الأولي سيحولها إلى كتلة شيفرة لتقييمها.

وإذا نظرت إلى كل من أمري RAED و GET..@ فسترى أن المعالج الأولي يحول كلاً من أمر GET..@ إلى عملين رئيسين هما: (أ) أنشيء هدف GET ، و (ب) أضفه إلى

مصفوفة قائمة GETLIST. ويتم عندئذ ترجمة أمر READ إلى استدعاء لوظيفة READMODAL( ) وتمرر له المصفوفة GETLIST.

كما أن المنطق يخدم هنا بشكل جيد. فيعمل أمر PROMPT @ الآن على إنشاء "object" قائمة الاختيارات ، ويضيفه إلى المصفوفة MENULIST. وقائمتنا هنا هي عبارة عن مصفوفة "object" التي تبدو على الشكل التالي:

Element	Contents	TYPE
1	Row	N
2	Column	N
3	Prompt	C
4	Message	C
5	Action	B

لاحظ استخدام أعلاه كتلة معلم نتيجة (blockify result-marker) لتحويل عبارة ACTION إلى كتلة شيفرة code block.

كما أن الأمر MENU TO يستدعي وظيفة LiteMenu( ) ويوصل إليها محتويات المصفوفة MENULIST ، ويوصل إليها ، مع هذه القائمة المتغيرين التاليين:

- عنوان الذاكرة للمتغيِّر الذي سيحتوي على الخيار الذي تم اختياره. وسيسمح هذا لوظيفة LiteMenu( ) من بلورة قيمته مباشرة.
- اسم المتغيِّر. ويطلب هذا في حالة كون وظيفة SET KEY تم بدؤها من داخل وظيفة LiteMenu( ) في حالة إنتظار. ويجب الانتباه إلى كيفية استخدام السلسلة الصامته لمعلم النتيجة (dumb stringify result-marker) لتحقيق هذا الأمر.

ربما أننا ألمعنا إلى "حالة انتظار" فهل حاولت يوماً ربط المساعدة الحساسة للمحتوى مع كل خيار من خيارات قائمة الاختيارات في MENU TO ؟ فإذا فعلت ذلك فقد تعلمت الطريقة الصعبة التي يمكن اعتبارها شبه مستحيلة. ولحسن الحظ ، فمن السهل تصميم وظيفة LiteMenu( ) بحيث تسمح بمستوى أعلى من هذه الوظيفة. فمثلاً ، إذا كنت تحفظ اختيار قائمة الاختيارات إلى SEL :

```
MENU TO SEL
```

وكنت في الخيار الثالث من قائمة الاختيارات ، فإن "SEL[3]" ستمرر إلى إجراء "المفتاح الساخن " Hot-key (انظر وظيفة SETKEY( ) في شيفرة المصدر لتلاحظ مدى سهولة تحقيق هذا الأمر). ولابد من القيام بأمرين اثنين لاستخدام وظيفة LiteMenu( ).

١- تأكد من تضمين ملف الترويسة الذي يحتوي على نسخ معدَّله من كل من أمري @..PROMPT و MENU TO.

٢- أعلن MENULIST في أعلى وظيفتك ، ويفضَّل أن تكون LOCAL .

```
/*
   Function: LiteMenu()
   Author:
      Dialect:  Clipper 5.01
   Compile:  clipper litemenu /n /w
   Purpose:  Preferred alternative to Clipper's @..PROMPT and MENU commands
*/

#include "inkey.ch"
#include "box.ch"
#include "litemenu.ch"

#define TEST            // manifest constant to compile test code
#ifdef TEST            // begin test code

function main
local menulist
local nsel := 1
set key 28 to helpme
scroll()
setcolor('w/b, n/bg')
dispbox(6, 32, 16, 47, B_DOUBLE + ' ')
@ 7, 35 say "Sample Menu" color "+gr/b"
@ 8, 32 say chr(199) + replicate(chr(196), 14) + chr(182)
do while nSel != 0 .and. nSel != 7
   @  9,33 prompt padr('Customers', 14) message 'Add/edit customer data' ;
         action CustFile()
   @ 10,33 prompt padr('Invoices ', 14) message 'Add/edit invoice data' ;
         action InvFile()
   @ 11,33 prompt padr('Vendors', 14) message 'Add/edit vendor data' ;
         action VendorFile()
   @ 12,33 prompt padr('Reports', 14) action Reports()
   @ 13,33 prompt 'reconci~Liation' action Reconcile()
   @ 14,33 prompt 'Maintenance    '  message "Rebuild indices, backup, etc." ;
         action Maint()
```

```
  @ 15,33 prompt padr('Quit', 14)
  menu to nSel
enddo
return nil

//------- stub functions for each menu option

static function CustFile
Output("You selected the Customers option")
return nil

static function InvFile
Output("You selected the Invoices option")
return nil

static function VendorFile
Output("You selected the Vendors option")
return nil

static function Reports
Output("You selected the Reports option")
return nil

static function reconcile
Output("You selected the Reconciliation option")
return nil

static function Maint
Output("You selected the Maintenance option")
return nil

static function HelpMe(p, l, v)
Output("You pressed F1...  Procedure: " + p + "    Variable: " + v)
Output("Grumpfish Library features excellent help screen development!")
return nil

static function Output(cMsg)
@ maxrow(), 0 say padc(cMsg, maxcol() + 1) color "+gr/r"
inkey(0)
scroll(maxrow(), 0)
return nil

#endif                  // end test code


//------- these manifest constants are for easy identification
//-------- of levels in the multi-dimensional array
#define  ROW            1
#define  COL            2
```

```
#define  PROMPT       3
#define  MESSAGE      4
#define  ACTION       5

/*
  LiteMenu() -- alternate menu system
*/
function LiteMenu(aOptions, nSel, cVarname)
local nElements := len(aOptions)
local nX
local nKey := 0
local cTriggers := []
local lFallout := .f.
local lOldmsgctr := set(_SET_MCENTER, .T.)
local nPtr
local nMessrow := set(_SET_MESSAGE)
local lOldcursor := SETCURSOR(0)
local cOldcolor := SETCOLOR()
local cPlainclr
local cHilitclr

//----- if MESSAGE row was never set, use the bottom row of screen
if nMessrow == 0
  nMessrow := maxrow()
endif

//----- set default colors for unselected and selected options
nX := at(',', cOldcolor)
cPlainclr := substr(cOldcolor, 1, nX - 1)
cHilitclr := substr(cOldcolor, nX + 1)

//------ determine initial highlighted item default to 1 -- also perform
//----- error-checking to ensure they didn't specify an invalid selection
if nSel == NIL .or. (nSel < 1 .or. nSel > nElements)
  nSel := 1
endif

//------ build the string containing available letters for nSel
for nX = 1 to nElements
  //---- the default is to add the first non-space character.
  //----- However, if there is a tilde embedded in this menu
  //------ option, use the letter directly following it.
  if (nPtr := at("~", aOptions[nX, PROMPT]) ) > 0
    cTriggers += upper(substr(aOptions[nX, PROMPT], nPtr + 1, 1))
  else
    cTriggers += upper(left(aOptions[nX, PROMPT], 1))
  endif
```

```
  ShowOption(aOptions[nX], cPlainclr)
next

//------- commence main key-grabbing loop
do while nKey != K_ENTER .and. nKey != K_ESC
  setcolor(cPlainclr)
  //---- display current option in highlight color
  @ aOptions[nSel, ROW], aOptions[nSel, COL] SAY ;
          strtran(aOptions[nSel, PROMPT], "~", "") color cHilitclr
  //----- display corresponding message if there is one
  if aOptions[nSel, MESSAGE] == NIL
    SCROLL(nMessrow, 0, nMessrow, maxcol(), 0)
  else
    @ nMessrow, 0 SAY padc(aOptions[nSel, MESSAGE], maxcol() + 1)
  endif
  if lFallout
    exit
  else
    nKey := inKey(0)
    do case

      /*
        use SETKEY() to see if an action block attached to the last
        keypress -- if it returns anything other than NIL, then you
        know that the answer is a resounding YES!
      */
      case setkey(nKey) != NIL
        /*
          pass action block the name of the previous procedure,
          along with the name of the variable referenced in the
          MENU TO statement and the current highlighted menu
          option (this means that you can tie a help screen to
          each individual menu option; try that with MENU TO)
        */
        eval(setkey(nKey), procname(1), procline(1), cVarname + ;
               "[" + ltrim(str(nSel)) + "]")

      //------ go down one line, observing wrap-around conventions
      case nKey == K_DOWN
        ShowOption(aOptions[nSel], cPlainclr)
        if nSel == nElements
          nSel := 1
        else
          nSel++
        endif

      //----- go up one line, observing wrap-around conventions
      case nKey == K_UP
```

```
        ShowOption(aOptions[nSel], cPlainclr)
        if nSel == 1
          nSel := nElements
        else
          nSel--
        endif

      //------ jump to top option
      case nKey == K_HOME
        //------- no point in changing color if we're already there
        if nSel != 1
          ShowOption(aOptions[nSel], cPlainclr)
          nSel := 1
        endif

      //------ jump to bottom option
      case nKey == K_END
        //---- no point in changing color if we're already there
        if nSel != nElements
          ShowOption(aOptions[nSel], cPlainclr)
          nSel := nElements
        endif

      //------ first letter - jump to appropriate option
      case upper(chr(nKey)) $ cTriggers
        ShowOption(aOptions[nSel], cPlainclr)
        nSel := at(upper(chr(nKey)), cTriggers)
        lFallout := .t.

    endcase
  endif
enddo
//------ if there is an action block attached to this nSel, run it
if lastkey() != K_ESC
  if aOptions[nSel, ACTION] != NIL
    eval(aOptions[nSel, ACTION])
  endif
else
  nSel := 0                // since they Esc'd out, return a zero
endif
setcursor(lOldcursor)
set(_SET_MCENTER, lOldmsgctr)   // reset SET MESSAGE CENTER
setcolor(coldcolor)
return nil

/*
  Function: ShowOption()
```

```
 Purpose:  Display current prompt in mixed colors
*/
static function ShowOption(item, cPlainclr)
local nPtr := at("~", item[PROMPT])
if nPtr > 0
  @ item[ROW], item[COL] say STRTRAN(item[PROMPT], "~", "") color cPlainclr
  @ item[ROW], item[COL] + nPtr - 1 say ;
              substr(item[PROMPT], nPtr + 1, 1) color '+'+cPlainclr
else
  @ item[ROW], item[COL] say left(item[PROMPT], 1) color '+'+cPlainclr
  dispout(substr(item[PROMPT], 2), cPlainclr)
endif
return nil

//------ end of file LITEMENU.PRG
```

الإعلانات المحلية والساكنة

اعتبر كليبر قبل الإصدار 5.x مجمّعاً بصورة رئيسة (ومنافساً أيضاً) لقاعدة البيانات dBASE III Plus ، وهكذا فقد نهجت لغة كليبر منهج قواعد البيانات لأسباب المنافسة لاغير. وقد أفلح هذا الاتجاه ، وأفاد في اكتساب انتقاد كبار المبرمجين بلغة قاعدة البيانات وخبرتهم في اكتشاف العيوب التي يحتويها البرنامج ، والذي أفاد بدوره من هذه الانتقادات الجيدة في إثراء اللغة وتطويرها بشكل مستمر.

إلا أن منافسة قواعد البيانات جلبت معها عدداً من المحاذير الظاهرة ، أحدها وأهمها هو عدم احتواء البرنامج على طريقة ثابتة لنقل (scoping) المتغيرات. كما أن الإعلانات الوحيدة الممكنة للمتغيرات في لغة قاعدة البيانات dBASE هي "العام" و"الخاص" أي "private" "public". ولحسن الحظ ، فإن كليبر 5.x يقدم متغيرين جديدين للإعلانات وهما: "STATIC" و "LOCAL". وتمكنك هذه الخيارات من جعل برامجك أسرع ، وأقل استخداماً للذاكرة وسهلة الصيانة إلى حد كبير.

جدول الرموز Symbol Table

عند الإشارة إلى متغيرات شديدة التغير مثل (PUBLIC/PRIVATE) يجب أن يتبّع برنامجك عملية ذات خطوتين لاشتقاق قيمتها. أولاً : يجب أن يبحث عن اسم هذا المتغير في جدول الرموز والذي يتم إنشاؤه أثناء تجميع برنامجك. ويحتوي جدول الرموز على عناوين ذاكرة يتم تخزين قيم المتغيرات فيها. وبعد أن يقرر برنامجك عنوان الذاكرة يمكنه أن يحدد قيمة المتغير.

إن استخدام جدول الرموز له حسنة واحدة ، إلا أنه له سيئتين:

- الحسنة : يتيح لك إمكانية استخدام عامل الماكرو لاستبدال متحولات الماكرو أثناء وقت التنفيذ وإن مثل هذا الاستبدال مستحيل دون وجود مُدخَلٍ في جدول الرموز.

- أما السيئة فهي : سيصبح الأداء بطيئاً بشكل ملحوظ جداً وذلك لأن كل إشارة مرجعية إلى متغير عام أو خاص ستكون عملية ذات مرحلتين.
- وأما السيئة الثانية فهي : سيكلفك كل متغير عام أو خاص في جدول الرموز حوالي ١٦ بايت . وإذا ضربت هذا الرقم بعدد المتغيرات العامة الموجود في أي برنامج من البرامج التي تعدَها أو تستخدمها (وهي مئات أوآلاف تقريباً). فإذا استخدمت (٥٠٠) خمسمئة متغير في برنامج ما فإن هذا سيضيف قرابة ٨ كيلو بايت من الوزن الميت الذي لا داعي له إلى ملفك التنفيذي . ولا شك أن كل المبرمجين والمستخدمين يحاولون جهدهم تقليص احتياجات الذاكرة كلما أمكنهم ذلك ، ولعل جدول الرموز هو أحسن مكان نبدأ فيه تقليص احتياجات الذاكرة.

لذلك نرى أن السيئات تفوق الحسنات بكثير ، كما أنه يمكن تجنب استبدال الماكرو بصورة عامة لأن هناك عدداً من الحلول البديلة التي يمكن استخدامها لمواجهة مثل هذه المشاكل لدى استخدام كليبر.

إن هناك ست عبارات متغيرات إعلان في كليبر وهي : خاص private ، عام public، محلي local،ساكن static ، حقل field ، متغير ذاكرة memvar. (كما يمكن اعتبار عبارة المعالم PARAMETERS أحد هذه الإعلانات أيضاً إذ أنها هي التي توضح بدء المتغيرات الخاصة PRIVATE).

فليسقط كل من إعلانات PRIVATE و PUBLIC

إنه لا مجال لأي من هذين الإعلانين في برنامج مكتوب بطريقة جيدة باستخدام كليبر. إذ أن لكل منهما جدول رموز ومدخلاته ، وهذا يعني أنهما سيعملان ببطء ويزيدان حجم العبء الذي تحمله (كما ذكرنا آنفاً). وعلاوة على هذين الأمرين فإن مجال رؤية كل من متغيرات "العام" و "الخاص" قد يقود إلى وجود عيوب خفية في البرامج يمكن أن تتطلب ساعات (وربما أياماً) لاكتشافها وتصحيحها. وإن هذين الإعلانين من السهولة بمكان استخدام مايسمى: "متغيرات موروثة" (وهي المتغيرات التي يمكن مشاهدتها في الوظائف

الدنيا ، دون أن يكون قد تم تمريرها كمتغيرات رسمية). وتعتبر هذه طريقة سيئة في البرمجة، وقد تودي بك إلى المشاكل دوماً . فإذا أردت كتابة وظيفة ما باستخدام كليبر تكون قابلة للاستعمال بشكل حقيقي في أية حالة من الحالات ، فيجب أن تقبل تلك الوظيفة قائمة رسمية بالمتغيرات فقط . فالوظيفة الحقيقية يجب ألا تفترض أية افتراضات مهما كان الأمر و السبب.

وإنك – عزيزي المبرمج والقارئ – حر ، بل لك مطلق الحرية لتفعل ما تشاء في برامجك إلا أننا نقترح اقتراحاً أكيداً أن تتجنب استخدام كل من خياري إعلانات "العام" "الخاص" فوراً ، كي لا تندم لاحقاً على ما قدمت. وإنك ستوفر بذلك على نفسك الكثير من الوقت والجهد اللازمين لاكتشاف الأخطاء البرمجية وتصحيحها (أو القلق الذي يساورك للتعرف على كيفية تحميل برنامجك بحيث يمكن تشغيله على شبكة محلية).

فليسقط كل من إعلانات FIELD و MEMVAR

يجب أن يتم تجنب استخدام كل من هذين الإعلانين أيضاً Memvar و Field كسابقيهما "العام" و "الخاص" ، فإنك باستخدام هذه الإعلانات لن تجهز "متغيرات" كما هي الحال لدى استخدام "الإعلانات" الأخرى ، بل إنك بدلاً من ذلك ، تخبر "المجمع" (برنامج التجميع كليبر) أن يفترض اعتبار الاسماء التي أدرجتها إما على أنها "حقول" ، أو على أنها "متغيرات ذاكرة".

إن إعلان "الحقل" هو علة في طريق البرمجة الجيدة ، إذ أنه يشجع المبرمج أن يكون كسولاً، بحيث أن يسبق اسم كل حقل من حقول قاعدة البيانات بالاسم المطابق له . كما أن إعلان " متغير الذاكرة " مفيد لكل من حالتي المتغيرات "العام" و "الخاص" ، وبما أننا اقترحنا عدم استخدام هذه الإعلانات ، فمن العبث محاولة استخدام هذين الإعلانين "حقل" و "متغير ذاكرة".

إعلان "محلي" LOCAL

إن هذه المتغيرات لا يمكن مشاهدتها إلا ضمن الوظيفة التي أعلنت من أجلها. ومع أن كلاً من هذين الإعلانين private و local قد يبدو مشابهاً للآخر في الوهلة الأولى ، إلا أن الفارق الكبير بينهما هو أن إعلان "محلي" لا يمكن مشاهدته في الوظائف ذات المستوى الأدنى.

وكذلك ، فإنك إذا أعلنت متغيراً ما على أنه "محلي" LOCAL ، إلا أنك لم تستهله فإن ذلك المتغير ستعين له قيمة الابتداء على أنها صفر (٠). وكذلك الحال ، فإنك إذا أعلنت مصفوفة محلية LOCAL array دون استهلالها فإن كافة عناصرها ستعطي قيمة الصفر (٠) أيضاً.

وعلاوة على هذا ، فإن الإعلان العلني لمتغيرات "محلية" local ، تنشأ أيضاً عندما تمرر متغيرات معينة إلى وظيفة ما باستخدام القائمة اللغوية syntax list بدلاً من استخدام عبارة PARAMETERS. وسيعامل كل من المتغيرين "A" و "B" في المثال التالي على أنهما متغيرات "محلية".

```
function myfunc(a, b)
```

- وقد يكون هذا التركيب اللغوي مزعجاً بعض الشيء إذا لم تكن قد اعتدت على استخدامه سابقاً. إلا أن هناك عدة أسباب تفرض عليك استخدام هذا التركيب اللغوي واجتناب استخدام عبارة PARAMETER ومن هذه الأسباب ما يلي:
- إن استخدام خيار Local بدلاً من Private يعني سرعة في التنفيذ وصغراً في حجم الملف التنفيذي ، وذلك بسبب تصغير "جدول الرموز".

- وبعد أن تتعود على استخدام هذا التركيب اللغوية المقترح ستجد أن القاعدة اللغوية للإعلان الرسمي أكثر مررونة وسهولة في الاستخدام. وسيمكنك أن تستنبط بنظرة سريعة ما هي المتغيرات التي تقبلها وظائفك.

مجال المتغيرات المحلية LOCAL

يبين المثال التالي مجال المتغيرات المحلية local:

```
function myfunc1
local mvae := 200
 myfunc2 ( @mvar )                        // pass by reference
? mvar                                     // 400
return nil

function myfunc2(mvar )
mvar *= 2
return nil
```

وستعلن الوظيفة ( )MYFUNC1 المتغير MVAR على أنه "محلي" ، ثم تستدعي الوظيفة ( )MYFUNC2 وتمرر الوظيفة MVAR بالإشارة. ومن الضروري القيام بذلك لأن المتغير MVAR لن يكون مشاهداً من قبل ( )MYFUNC2. وهناك احتمال آخر يمكن القيام به هنا وهو أن تحمل ( )MYFUNC2 القيمة الراجعة ، ثم يتم تعيينها بعد ذلك للمتغير MVAR.

ملاحظات عن المتغيرات المحلية

- يمكن تعيين المتغيرات "المحلية" في الوقت الذي يتم إعلانها فيه باستخدام عامل التحديد السطري.

  ويجب الانتباه إلى عدة أمور أثناء تعيين متغيرات "محلية". أولا:

  إذا استخدمت التركيب اللغوي التالي لتأسيس ثلاث متغيرات محلية Local :

```
Local  x := y := z := 0
```

فإن متغير X فقط هو الـذي سيكون محلياً LOCAL. أمـا المتغـيران Y و X فسيعاملان على أنهما متغيران خاصان PRIVATE . وإن الطريقة المناسبة لكتابة مثل هـذا الـتركيب اللغوي هي ما يلي:

```
localo  x := 0,  Y := 0,  Z := 0
```

وأما النقطة الثانية ، فهي أنك لا تستطيع استخدام المتغير "المحلي" لتعيين متغير محلي آخر في العبارة ذاتها. وقد يتم تجميع هذا البرنامج (المشاهد أدناه) دون مشاكل تذكر إلا أنه يجـب الانتباه التام أثناء تشغيل البرنامج و تنفيذه.

```
function  pasword(string )
local  x  := len(string) ,  midpoint := int((maxcol( ) - x )  /  2 )
```

والمشكلة هنا هي أن تعيين "نقطة وسط" MIDPOINT يعتمد على قيمة المتغـير X ، ولا يمكن أن يتم هذا في العبارة التي تم فيها تعيين المتغير X ذاته.

- لذلك ، فالحل لمثل هذه المشكلة أن تقسم هذه العبارة إلى عبارتين ذات متغـير "محلـي" LOCAL ، كما يلي:

```
function  pasword(string )
local  x  := len(string )
local  midpoint :=   int( ( maxcol(  ) - x )  /  2 )
```

بل إننا نقترح في الواقع أن تخصص عبارة مستقلة لكل إعلان للمتغير "المحلـي" بشـكل مستقل على حدة حتي يتم التوضيح بشكل كامل.

- عند إعلان متغير "محلي Local" في وظيفة ما ، يجب أن يسبق هذا الإعلان أيـة عبـارة تنفيذيـة (قـد تتضمـن أيـاً مـن المتغـيرات التاليـة: Private و Public أو حتـى Parameters). ولـن يتـم تجميـع البرنـامج في المثـال التـالي لأن عبـارة متغير Private سبقت متغير Local ، كما في المثال أدناه:

```
function myfunc
private whatever := 100
local mvar := 200
return nil
```

■ إن المتغيرات المحلية Local تخفي كلاً من المتغيرات العامة Public والخاصة Privats وأي حقل قاعدة بيانات يحمل الاسم ذاته. إلا أنها لاتلغي المتغيرات أو الحقول التي تم إعلانها باستخدام إعلانات "متغير الذاكرة" MemVar أو الحقل Field (والتي يجب ألا تستخدمها على أي حال ، كما يبين المثال التالي:

```
function main
private mvar : = 5
test ( )
? mvar          // still  5
return nil

function test
local mvar : = 1
? mvar          // 1, not 5
return nil
```

■ لايمكن استبدال المتغيرات المحلية Local باستخدام ماكرو ، وذلك لما ذكرناه سابقاً بأنها ليس لها مدخلات في "جدول الرموز".

■ ويجب استخدام الوظيفة VALTYPE( ) بدلاً من استخدام الوظيفة TYPE( ) لاختبار نوع المتغير المحلي Local ، إذا أن الوظيفة TYPE( ) لايمكن أن تعمل إلا على العناصر التي لها مدخلات (قيم) في "جدول الرموز". وبما أن المتغيرات المحلية ليس لها قيم في "جدول الرموز" فلن يكون أي تأثير للوظيفة TYPE( ) عليها ، كما في المثال التالي:

```
function main
local mvar := 1
? type( 'mvar' )                    // "U" (undefined )
? valtype(mvar)                     // "N" ( unmeruc )
return nil
```

■ لايمكن حفظ المتغيرات المحلية Local أو استرجاعها من ملفات الذاكرة. MEM وهذا كذلك للسبب ذاته الذي أشرنا اليه آنفاً أنه ليس لها قيم في "جدول الرموز".

ويجب الانتباه الدقيق لهذه الحقيقة عند محاولة تحويل كل المتغيرات الخاصة Private إلى متغيرات محلية Local . وسنبين خلال حديثنا عن المصفوفات Array كيف يمكن حفظ المعلومات الإجمالية دون استرجاع ملف ذاكرة MEM.

- على خلاف كل المتغيرات Public و Private (والتي هي محددة بالعدد ٢٠٨٤ متغير) فليس هناك حد للعدد الإجمالي لمتغيرات Local ضمن برنامجك، فيمكنك أن تضع أي عدد من المتغيرات المحلية.

الإعلان الساكن STATIC

يشبه المتغيرات الساكنة STATIC ، فيما يتعلق بمشاهدتها ضمن الوظيفة التي تعلنها فيها إلا أنها تختلف عن المتغيرات المحلية لأنها تحتفظ بقيمتها خلال فترة تنفيذ البرنامج كلها. ولعل هذا المفهوم غريب على كثير من مطوري البرامج ، وسنحاول أن نبينه من خلال المثال التالي:

```
function main
for x := 1 to 1000
   ? counter( )
next
return nil

function counter
static y := 0
return ++y
```

وسنبين أنه في كل مرة تنفذ فيها حلقة FOR...NEXT في الوظيفة Main( ) ستزداد القيمة الراجعة باستخدام وظيفة العداد Counter( ). أي ، بمعنى آخر: في المرة الأولى التي يتم فيها استدعاء وظيفة العداد Counter( ) ستصبح قيمة Y= صفراً ، وستتم زيادة هذه القيمة قبل تشغيلها إلى ١ ، والتي هي القيمة الراجعة من قبل وظيفة العداد Counter( ). وأما في المرة الثانية لاستدعاء العداد ستحتفظ Y بقيمتها السابقة أي (١) وتصبح الوظيفة (٢) ، وهكذا.

والسبب في ذلك أنك عند تأسيس المتغير الساكن STATIC. فإن هذا السطر من البرنامج ستتم معالجته أثناء وقت التجميع ، وليس أثناء وقت التشغيل. وسيكون هذا سريعاً إذا ما قورن بإعلانات كل من متغير Public و Private إذ أن هذه تحتاج إلى إعادة تأسيس في كل مرة تدخل فيها الوظيفة ، بل إذا أعلنا مثلاً متغير Y على أنه Local أو Private ، فإن قيمته ستعود إلى "1" في كل مرة نستدعي فيها العداد ( )Counter ، والتي ستعود إلى " < " دائماً ، وهذا أبعد مايكون عما قصدناه.

ويبدو هذا الأمر غير عادي لأننا اعتدنا أن يتم تنفيذ كل سطر من سطور البرنامج في كل مرة ندخل وظيفة ما. إلا أنك إذا اتبعت خطوات عمل البرنامج باستخدام برنامج اكتشاف الأخطاء وتصحيحها Debugger ستلاحظ أن هذا البرنامج سيتجاوز إعلانات المتغيرات الساكنة STATIC ، لأنها ليست رموزاً تنفيذية . ولايعتبر إعلان متغير ساكن STATIC ظاهر الأهمية إلا في وقت التجميع فقط.

وكما هي الحال مع متغير Local فإنك إذا أعلنت متغيراً ساكناً STATIC دون تأسيسه فسيتم إعطاء هذا المتغير قيمة الصفر (0) ، وعلى غرار هذا فإن أية عناصر في مصفوفة ساكنة غير مؤسسة سيتم تأسيسها بشكل آلي لتأخذ القيمة صفر (0) ذاتها أيضاً.

مجال المتغيرات الساكنة

سنبين في المثال التالي مجال اثنين من المتغيرات الساكنة:

```
function main
static y :=  " testing "
for x := 1 to 100
   ? myfunc( ) , y
next
return nil


function  myfunc
static y :=  100
return  --y
```

إن خيار ( )Main يعلن متغير Y على أنه متغير ساكن ، ويقوم بتنفيذ حلقة FOR...NEXT التي تستدعي وظيفة ( )MyFunc. كما أن وظيفة ( )MyFunc تعلن المتغير Y على أنه متغير ساكن ، وتوسسه على القيمة (١٠٠) أثناء وقت التجميع. وستزيد وظيفة ( )MyFunc في كل مرة يتم استدعاؤها فيها قيمة Y وترجع تلك القيمة. وبما أن Y هو متغير ساكن ستحتفظ بقيمتها للمرة التالية التي يتم استدعاء وظيفة **( )MyFunc** فيها ، وتجدر الإشارة إلى الأمر الهام هنا وهو أن نسختي المتغير Y ستكونان مشاهدتين ضمن وظيفتيهما المحددتين.

ملاحظات على المتغير الساكن STATIC

- وكما هو الحال في المتغيرات المحلية ، فعندما تعلن عن متغير ساكن static في وظيفة ما يجب أن يسبق أية عبارات تنفيذية (والتي تشمل كلا من المتغيرات Private و Public و Parameters). ولن يتم تجميع البرنامج التالي لأن Public سبقت العبارة STATIC ، كما في المثال التال :

```
function counter
public mvar := ' test '
static mcounter := o
return ++mcounter
```

كما يمكن أن تعلن متغيرات Static أيضاً قبل عبارة الوظيفة الأولى أو الإجراء الأول في ملف برنامج PRG. وسنبين هذا لاحقاً.

- تحتل كل من المتغيرات Static و Local الأولية قبل كل من متغيرات private , public وأي حقل من حقول قواعد البيانات تحمل الاسماء ذاتها. إلا أن هذه المتغيرات لا تلغي المتغيرات أو الحقول التي تم إعلانها باستخدام أي من متغيرات Memver أو متغير Field.

- لايمكن استبدال متغيرات STATIC بماكرو ، كما هي الحال في متغيرات LOCAL ، إذ أنها لا توجد لها قيم في "جدول الرموز".

- يجب استخدام خيار UALTYPE( ) بدلاً من خيار TYPE( ) لاختبار متغير STATIC ، كما أشرنا سابقاً ، إذ أن خيار TYPE( ) يعمل فقط إذا كان للمتغير قيمة في " جدول الرموز " . وبما أن كلاً من متغيرات STATIC وLOCAL ليس لها قيمة في " جدول الرموز " سيكون نوعها دائماً "غير محدد" (Undefined) "U".
- لايمكن حفظ متغيرات STATIC أو استرجاعها من ملف ذاكرة (.MEM).
- على خلاف كل من متغيرات PUBLIC و PRIVATE ليس هناك أية حدود للعدد الإجمالي من المتغيرات الساكنة STATIC ضمن أي برنامج من البرامج.

المتغيرات الساكنة على مدى الملف

إذا أعلنت متغيرات ساكنة قبل عبارة أية وظيفة Function أو إجراء Procedure سيصبح مجال هذه المتغيرات على مدى الملف بكامله. وتصبح مشاهدة من قبل جميع وظائف ذاك البرنامج. ويمكن اعتبار هذه المتغيرات الساكنة على أنها متغيرات عامة محدودة "limited public" ، وهي مشابهة للمتغيرات العامة ، ألا أنها للملف ذلك البرنامج فقط.

ويبين المثال التالي مبدأ (وقوة) المتغيرات الساكنة على مدى الملف جميعه. ولدينا هنا ملفان لبرنامجين هما: SCRNSAVE.PRG و MAIN.PRG .

```
/* MAIN.PRG_must be compiled with /N option */
function main
@ 0, 0, maxrow( ) , maxcol( ) box replicate ( " * ", 9 )
gsavescrn( )
inkey (2)
croll ( )
@ 12, 20 say " you are staring at a mostly empty screen"
@ 13, 20 say " press any key to restore previous screen"
inkey (o)
grestscrn( )
return nil

* eof main.prg

/* ------------------------ */
```

```
/* SCRNSAVE.PRG-must be compiled with /N option */
static buffer

function grestscrn(t , l , b , r)
/* establish default window parameters if not passed */
t := if( t == NIL, 0, t )
l := if( t = = NIL, 0, t )
b := if( t == NIL, maxrow ( ) , b )
r := if ( t = = NIL, maxcol ( ) , r )
buffer := { t , l , b , r savescreen( t, l , b, r ) }
return nil

function grestscrn
restscreen( buffer[1] , buffer[2] , buffer[3] , buffer[4] , buffer[5] )
return nil
* eof scrnsave.prg
```

إن الوظيفة ( )Main تملأ الشاشة بنجوم جميلة تستدعي الوظيفة ( )GSaveScrn وهي وظيفة يحتوي عليها برنامج يسمى SCRNSAVE.PRG ، وينشيء هذا البرنامج مصفوفة من خمسة عناصر ، تتطابق العناصر الأولى منها مع أعلى الشاشة ، ويسارها ، ويمينها ، وأسفلها ، وإحداثيات مناسبة لذاكرة الشاشة التي يراد حفظها. وأما العنصر الخامس والأخير فيحتوي على المحتويات الحقيقية من ذاكرة الشاشة. وتحفظ هذه المصفوفة في متغير ساكن على مدى الملف يسمى BUFFER والذي لا يمكن مشاهدته سوى من خلال الوظيفة ( )GSaveScrn والوظيفة ( )GRestScrn. وإذا حاولنا التوصل إلى هذه الذاكرة المؤقتة BUFFER من الوظيفة ( )Main فإن البرنامج سيتوقف ويتحطم بسرعة هائلة.

ويعود التحكم إلى الوظيفة ( )Main والتي تمسح الشاشة وتعرض رسالة وتنتظر منك أن تضغط على مفتاح من المفاتيح على لوحة المفاتيح. وتعمل الوظيفة ( )Main على استدعاء الوظيفة ( )GRestSern لاسترجاع الشاشة كما كانت عليه من قبل. وتشير هذه الوظيفة إلى مصفوفة الذاكرة المؤقتة BUFFER على مدى الملف ، بحيث تجذب إحداثيات الشاشة بلطف من العناصر الأربعة الأولى ومحتويات الشاشة من العنصر الخامس.
والذي حققناه في هذا المثال البسيط هو عملية "الكبسلة" وهو أكثر من كلمة تقال..

الكبسلة Encapsulation

إن كليبر من البرامج التي يمكن تغيير شكل المتغيرات فيها من شكل إلى آخر. ومع أن هـذه الميزة قد تكون من المزايا الرائعة التي تتميز بها كليبر عن سائر لغات البرمجـة الأخـرى ، إلا أنها قد تكون من الأمور المزعجة جداً أحياناً ، ويبـين المثال التـالي كيـف يمكـن أن يسبب تغيير شكل متغيرات البيانات شيئاً من الإزعاج:

```
function whatever
private  x
x : = 5
whatever2( )
x : = x * 5      // crashes because X is now a character string
return x

function whatever2
x : = "now I am a character string"
return nil
```

إن المتغيرات الساكنة الموجودة على مدى الملف جميعه تمكنك مـن "كبسـلة" البيانـات مـع الوظائف التي تريد التوصل إليها فعلاً فقط. وباستخدام طريقة "الكبسلة" يمكنـك أن تجعـل برامجك على شكل وحدات مترابطة وكذلك تمكنك من حـذف الأخطـاء البرمجيـة المزعجـة بأن تقلل من إمكانيات "الإيقاف" المؤقت للمتغيرات بحيث يتم الكتابة فوقها.

ويجب الانتباه إلى أن هذا التركيب يشكل نوع طبقة الهدف object class والـذي لايقـل في الغمـوض عـن وبحمـوع كـل مـن "البيانـات" و"الشـيفرة" ، فالبيانـات هنـا هـي الذاكرة المؤقتة للشاشة و الشيفرة التي تعمل عليها وتتألف من كل من وظيفـتي الحفـظ و الاسترجاع كما كان.

وعندما تزداد معرفتك عن المصفوفات التي يمكن تغيير حجمها بشكل ديناميكي ، يمكنـك استـخدامها مـع إعلانـات المتغـيرات السـاكنة علـى مـدى الملـف لإعـداد وظـائف وحدات "قابلة التطبيق فوق بعضها"stack-based" والتي تدهشـك بفعاليتهـا واختصارهـا للوقت وقدرتها.

أما الفائدة الأخرى لاستخدام إعلان متغير ساكن على مدى ملف بأكمله فهو حذف المتغيرات العامة من نوع PUBLIC من برامجك ، وسيمكنك هذا من تصغير حجم "جدول الرموز" (والملف التنفيذي EXE.) كما يمكنك من تنفيذ برامجك بسرعة أكبر. إلا أن الأهم من ذلك هو أن تحول دون تغيير المتغيرات العامة Public بالخطأ (أو لو قدر الله أن تحرر RELEASE).

متغير تحذير ساكن على مدى الملف

لعلك تذكر من قراءتك التي تقدمت في هذا الكتاب أنه إذا استخدمت المتغيرات الساكنة على مدى الملف فيجب أن تقوم بتجميع البرنامج باستخدام خيار N/ وكثير من المبرمجين يقعون في هذا الخطأ وينسون استخدام هذا الخيار ، ولاداعي لأن نُركز أكثر من هذا على هذا الموضوع ، فيجب الانتباه له تماماً ، واستخدام الخيار N/.

ولدى استخدام الخيار N/ ينشيء كليبر "إجراء بدء تشغيل ضمني لملف البرنامج المطلوب تشغيله" ، ويقوم هذا الإجراء بتعطيل دور المتغيرات الساكنة على مدى الملف تعطيلاً تاماً. ولنضرب لك مثالاً على هذا كما يلي:

```
/*   TEST . PRG   */

static marray_  : = { 'Lou' , 'Joe' , 'Paul' }

function func1
? marray_[1]
func2( )
return nil

function func2
? marray_[2]
func3( )
return nil

function func3
? marray_[3]
return nil
```

وسيعمل هذا البرنامج بشكل ممتاز إذا استخدمت خيـار ال/N ، أمـا إذا نسيت اسـتخدام هذا الخيار. فإن كليبر سينشيء "إجراء بدء تشـغيل ضمـني" يسـمى Test وسـيكون مجـال المصفوفة المسماة MARRAY محصوراً فقط على هذا الإجراء الوهمي المسمى TEST.

كما يجب الانتباه أيضاً إلى حالة أخرى وهي الإعلان المكرر. فلنفترض أنك أعلنت متغيراً محلياً Local. ثم قررت بعد ذلـك أن تجعـل هـذا المتغـير سـاكناً Static علـى مـدى الملف كله إذ يجـب أن يكـون مشـاهداً في وحـدات البرنـامج الأخـرى في ذلـك البرنـامج. فيجب أن تعلن أن ذاك المتغير ساكنٌ Static فوق أول وظيفة لذلك البرنـامج ولا تنسـى أن تزيل الإعلان المحلي Local ، إذ أنه إذا نسيت فإنه سيلغي المتغير الساكن ويسـبب لـك مختلف أنواع الإزعاجات. وتأمل المثال التالي:

```
static counter := 0

function main
local counter : = 1   // whoops ! forgot to delete this one
? counter              // 1-looking at the LOCAL
myfunc( )
return nil

function myfunc
? counter              // o-looking at the STATIC
return nil
```

ويمكنك أن تشاهد الاضطراب الكبير الذي سيحدثه لك هذا الأمر ، لذلـك يجـب الانتبـاه إلى ضرورة حذف عبارة المتغير المحلي Local فوراً إذا قـررت تغيـير متغـير محلـي إلى متغـير ساكن على مدى الملف.

تأسيس المتغيرات الساكنة/إعادة تجهيزها

يمكن تعيين متغيرات ساكنة في الوقت ذاته الذي يتم إعلانها فيه باسـتخدام عـامل التحديـد السطري in-line assignment ، إلا أنه يجب أن تسـتخدم الثوابـت البسـيطة لتحديدهـا. ولاتقبل استدعاءات الوظائف لأنه قـد تم تأسيس المتغـيرات السـاكنة قبـل وقـت التنفيـذ ،

وسنبين في المثال التالي كيف أن متغير الذاكرة MVAR لايمكن تأسيسه لأنه لايمكن تقييم وظيفة التاريخ ( )DATE. إذ أنك لم تشغل البرنامج بعد .

```
static mvar := date( )
```

إذا اضطررت لتعيين وظيفة ما على أنها متغير ساكن فيجب أن تقوم بذلك أثناء وقت التنفيذ بدلاً من أثناء وقت التجميع فإنك إذا فعلت ذلك سيكون أمامك خياران هما:

١) دمج الاختبار"NILTest" لتقرير ما إذا قد تم تأسيس المتغير الساكن STATIC أم لا وذلك على النحو التالي:

```
function counter
static y
if y == NIL
   y : = date( )
endif
return ++y
```

وإن المحذور الواضح لهذه الطريقة هو أنك ستواجه جزاء الأداء عند تنفيذ العبارة الشرطية IF في كل مرة تالية تستدعي فيها تلك الوظيفة.

٢) اجعل المتغير ساكناً على مدى الملف ، ثم اكتب وظائف مستقلة للتوصل إليه وتأسيسه، وإن استخدام هذه الطريقة سينتج برنامجاً شبيهاً بما يلي:

```
STATIC mvar

function initcount
mvar := date( )
return nil

function counter
return mvar++
```

ولاشك أن الطريقة الثانية هي أفضل بكثير من الطريقة الأولى (لاحظ أيضاً أنه يمكنك استخدام إعلان INI لتؤسس المتغيرات الساكنة من هذا النوع).

الوظائف الساكنة Static Functions

لاحظنا أن الإعلان الساكن يعطينا القدرة على "إخفاء" البيانات عن الوظائف الأخرى. ويمكنك أيضاً استخدام "السواكن" هذه لإخفاء الوظائف ذاتها عن الوظائف الأخرى. وإن إعلان وظيفة ما على أنها ساكنة يحدّ رؤيتها فقط بحيث لا ترى إلا من قبل الوظائف الأخرى الموجودة في ملف البرنامج ذاته. ويحد هذا إلى حد كبير من تعارض اسماء الوظائف.

ولعل أحد الأمثلة الرئيسة على تعارض الاسماء هو الوظيفة ( )Center ، إذ أن لكل مبرمج طريقته الخاصة في استخدام هذه الوظيفة. وبالتالي فإن التركيب اللغوي يختلف اختلافاً كبيراً ومتفاوتاً بحيث يسبب مثل هذا الاضطراب.

إلا أنك تستطيع الآن إخفاء وظائفك هذه عن وظائف الآخرين بإعلانها على أنها "ساكنة". ويمكن اعتبار المثال التالي في ملفي برنامجين مستقلين أحدهما هو "MAIN.PRG" والآخر هو "FUNCS.PRG" على النحو التالي:

```
/* MAIN.PRG */
function main
whatever( )
center ( 16,  "Ouch!" )        // run-time error
return nil

* eof main.prg
/* ---------------------- */
/* FUNCS.PRG */
function whatever
center( 17,  "this is a test" )
return nil


static function center( row, msg )
@ row, int ( ( maxcol ( ) + 1 - len ( msg ) ) / 2 ) say msg
return nil


* eof funcs.prg
```

فإن وظيفة ( )Center لن تكون مشاهدة إلا للوظائف والإجراءات الأخرى ضمن ملف برنامج FUNCS.PRG. فعندما تستدعي الوظيفة ( )whatever من البرنامج الرئيسي MAIN.PRG فلن تكون هناك أية مشكلة تواجهك عندما تستدعي هذه الوظيفة وظيفة ( )Center إذ أنهما كلاهما في البرنامج ذاته ، إلا أنك عندما تستدعي وظيفة ( )Center مباشرة من ملف البرنامج الرئيسي MAIN.PRG فسيتوقف البرنامج عن العمل فوراً.

ويمكنك باستخدام الوظائف الساكنة استخدام عدداً من الوظائف تحمل الاسم ذاته خلال برنامجك كله ، طالما أنها موجودة في ملفات برامج مستقلة.

وإذا كنت من المبرمجين الذين يعملون ضمن مجموعة مبرمجين فإنك ستستمتع جداً بالوظائف الساكنة وليس لأنها تحول دون تعارض الأسماء فقط ولكن لأنها تبسط توثيق البرامج أيضاً ، وتصور نفسك تعمل على وحدة من وحدات البرنامج تحتوي على عدد كبير من الوظائف.

```
function entry( a, b, c, d )
*
return nil
static function func1( )
*
return nil
static function func2( )
*
return nil
etsetera
```

وعندما يحين الوقت لتوزيع وحدتك من البرنامج على المبرمجين الآخرين فلن تحتاج إلا لتوثيق المتغيرات اللازمة للوظيفة ( )Entry فقط ، ولن يكون المبرمجون الآخرون بحاجة لمعرفة أي شيء عن الوظائف الساكنة والتي تستخدم فقط داخل ملف برنامجك الخاص ، فلن يكون هناك أي احتمال ، ولو بسيط ، أن تتضارب اسماء وظائفهم وتتعارض مع اسماء الوظائف التي سميتها أنت.

قاعدة عامة

عندما تريد تحديد ما إذا كانت وظيفة ما ستستخدم على أنها ساكنة : سل نفسك السؤال التالي : "هل أحتاج إلى استدعاء هذه الوظيفة من خارج ملف البرنامج الحالي؟" فإذا كان الجواب سلباً ، فيمكن أن تضع هذه الوظيفة على أنها وظيفة ساكنة Static.

تحذير

إذا استخدمت أياً من الوظائف التالية : تحرير مذكرة ( )MEMOEDIT أو الوظيفة ACHOICE() ، أو الوظيفة DBEDIT() باستخدام وظيفة يحددها المستخدم فلن يمكنك إعلان الوظيفة التي يحددها المستخدم على أنها ساكنة إذ أن هذه الوظائف الثلاث المذكورة تعتمد أساساً على "جدول الرموز". وإن الوظائف الساكنة ليس لها مدخلات/قيم في جدول الرموز العامة ، فيرجى الانتباه لهذه النقطة.

ولبين هنا على سبيل المثال وظيفة تم تحديدها من قبل المستخدم وارتبطت بالوظيفة ACHOICE() وهي تعالج كلاً من مفاتيح Enter و Esc و مفتاح المسافة Spacebar.

```
#include "inkey . ch"
#include "achoice . ch"
function main
local marray := { 'one' , 'tow' , 'three' } , ele
scroll ( )
ele := achoice( 11, 38, 13, 42, marray, .t. , 'MyFunc' )
return nil

statuc function myfunc( status , curr_elem , curr_row )
local key : = lastkey ( )
if key = K_ESC
  return AC_ABORT
elseif key = K_ENTER
  return AC_SELECT
elseif key = 32
 @ 20,0 say "You pressed spacebar ! Boy am I smart ! "
```

```
  inkey (0)
  scroll ( 20, 0, 20, 36, 0 )
endif
return AS_CONT
```

جمّع هذا البرنامج واربطه ، ثم اضغط على قضيب المسافات ، ولن يحدث أي شيء. ثم احذف كلمة static من إعلان MyFunc() ، وأعد تجميع البرنامج من جديد ، ثم حاول الضغط على قضيب المسافات مرة ثانية وانظر ماذا يحدث.

وظائف التأسيس INIT

تتمتع الوظائف التي أعلنت على أنها تأسيسية INIT بصفات متميزة جداً إذ أنها تنفذ فور البدء بتشغيل البرنامج ، وهذه الصفة تجعل هذه الوظائف مثالية لتحديد متغيرات ساكنة STATIC يجب أن تفترض قيمة ما تعود من قبل وظيفة ما.

ويبين البرنامج التالي استخدام وظيفة التأسيس INIT ، ومع أنها تظهر الوظيفة الأساسية Main( ) قبل وظيفة InitDate( ) فإن هذه الوظيفة سيتم تنفيذها أولاً ، وذلك لأنها تعيين تاريخ النظام إلى المتغير المسمى THEDATE.

```
static thedate          // file-wide

function test
? "TODAY's date : " , today ( )
//  . . .
return nil

init function initdate
thedate := date ( )
return nil

static function today
return thedate
```

إذا استخدمت وظيفة INIT في أكثر من ملف واحد في برنامجك فإنها ستنفذ فور بدء تشغيل البرنامج ، وفي هذه الحالة سيتم تنفيذ الوظائف بالترتيب الذي ربطت به بالبرنامج.

تحذير

إذا قررت استخدام وظائف INIT فيجب الانتباه إلى أنها ستنفذ قبل تنفيذ برنامج "معالج الأخطاء ERRORSYS.PRG" في كليبر ، وهذ يعني أنه إذا وقع خطأ ما في أي وظيفة من الوظائف فستحصل على رسالة خطأ لامعنى لها على الإطلاق.

ملاحظة هامة

عندما تسبق كلمة INIT الإعلان عن وظيفة Function أو إجراء Procedure فستنفذ هذه الوظيفة أو الإجراء قبل أول عبارة قابلة للتنفيذ في برنامجك. ويمكن استخدام مثل هذه الوظائف لاستهلال أهداف objects أو متغيرات ساكنة على عرض الملف ، ولفتح سجل وماشابهه.

وظائف الخروج Exit Functions

تعتبر هذه الوظائف نقيضة لوظائف التأسيس ويتم تنفيذها بعد تنفيذ آخر عبارة تنفيذية في برنامجك. وهذه الوظائف مفيدة في حفظ معلومات الإعداد والتهيئة ، وإنهاء جلسة عمل اتصالات ، وإغلاق ملف تسجيل عمليات وهكذا.

سير خطوات التحميل/الخروج من كليبر

عند البدء بتشغيل برنامج ، سيقوم كليبر بتنفيذ كل الخطوات التالية:

- تأسيس أي متغيرات ساكنة.
- استدعاء وظائف التأسيس INIT.
- استدعاء برنامج أخطاء النظام ERRORSYS( ) لتأسيس عمليات معالجة الأخطاء.

- استدعاء الوظيفة الأولى ، أو وظيفة الدخول.

ولدى الإنتهاء من العمل في البرنامج ، ستحدث الخطوات التالية:

- استدعاء أي وظيفة من وظائف الخروج EXIT .
- إغلاق أي ملف مفتوح تتم معالجته.
- مسح الذاكرة التخيلية (VM).
- الخروج إلى نظام التشغيل.

الوحدات البرمجية MODULARITY

بعد أن تعرفنا على استخدام المتغيرات وتطبيق نطاق المتغيرات بشكل واضح وأصبحنا جاهزين لاستخدامها ، فقد أصبحت برامجنا على شكل وحدات. وسنناقش في هذا القسم وظائف الوحدات المتوفرة في كليبر.

احتوت الإصدارات السابقة من كليبر على وسائل تمكّن المبرمج من حفظ ضوابط الألوان واسترجاعها كما كانت باستخدام الخيار (SETCOLOR) وكذلك خيار مكان المؤشر (ROW,COL). والشكل الموجود على الشاشة (SAVSCREN) إلا أننا لم نستطع آنئذ حفظ الضوابط العامة للبرنامج مثل كل من DECIMALS و SOFTSEEK وغيرهما كحجم المؤشر والمفاتيح "الساخنة".

ولحسن الحظ فقد أوجد الإصدار الجديد من كليبر حلاً ناجعاً لهذه المشاكل باستخدام الوظائف الجديدة التالية وهي : ( )SET و ( )ESTCURSOR و SETKEY( ) ، وسنتحدث عن كل منها بالتفصيل فيما يلي:

الوظيفة SET( )

تقبل هذه الوظيفة متغيرين هما:

- الأول : هو ساكن يمثل الضبط الذي تريد التساؤل عنه و / أمر تغييره. ابحث عن خيار SET.CH الموجود في البرنامج لتطلع على قائمة كاملة بسواكن البيان manifest constants و مصطلحات الاسماء سهلة التذكر جداً كما سترى في المثال التالي أدناه.

- أما المتغير التالي الاختياري فهو بمثابة قيمة تغيُّر الضبط إلى الحد الذي تريده. لاحظ المثال التالي أدناه ، والذي يضبط خيار DELETED ON.

ويمكنك أيضاً استخدام خيار منطقي "حقيقي" مع أنه متغير ثالث للوظيفة SET_ALTFILE و SET_PRINTFILE_ (والذي يقابل كلاً من أمري SET ALTERNATE TO و SET PRINTER TO على التوالي) وسيجعل هذا الأمر ملف الهدف مفتوحاً إذا وجد ، وهذا تحسين رائع على الإصدارات السابقة من كليبر بحيث يمكن إعادة تجهيز ملف الهدف من لا شيء.

الوظيفة SETCURSOR( )

تعمل هذه الوظيفة على غرار الوظيفة SETCOLOR( ) إلا أنها تتعامل مع حجم المؤشر المستخدم ، فإذا لم تمرر متغيراً فإنها تتضمن الحجم الحالي للمؤشر والذي يمكن أن يكون أحد خمسة أنواع هي كما يلي:

وصف المؤشر	الثابت الظاهر المشارك الموجود في ملف SETCURS.CH
لامؤشر (لايوجد)	SC_NONE
عادي (شرطة معترضة – وامضة)	SC_NORMAL
إقحام (نصف كتلة سفلية)	SC_INSERT
كتلة كاملة	SC_SPECIAL1
نصف كتلة عليا	SC_SPECIAL2

فإذا مررت متغيراً فإن SETCURSOR( ) ستغير شكل المؤشر إلى الحجم المطلوب طيلة مدة الانتظار حتى الإعادة إلى القيمة الحالية.

وظيفة ضبط المفتاح الساخن SETKEY( )

تشبه هذه الوظيفة وظيفة ( )SET القديمة ، وتمكّنك من تغيير حالة مفتاح ساخن لأية قيمة من قيم INKEY ، وهي تقبل متغيرين:

- المتغير الأول : هو رقمي لقيمة INKEY للمفتاح الذي يراد اختباره. وبدلاً من الإشارة إلى الأرقام مباشرة ، فإننا نقترح استخدام ثوابت البيان التي يحتوي عليها ملف INKEY.CH كما سنبين أدناه.
- المتغير الثاني الاختياري فيمكن أن يكون أحد شيئين (أ) كتلة شيفرة يمكن ربطها بعد ذلك بالمفتاح ذاته ويمكن تقييمها كلما ضغطت على المفتاح المحدد أو (ب) قيمة الصفر NIL والتي توقف عمل هذا المفتاح الساخن مباشرة. أما المبرمجون الذين لايرغبون استخدام كتلة الشيفرة ، فيمكنهم الاستمرار في استخدام طريقة أمر SET KEY لضبط حالة المفتاح. وسيبين المثال الموضح أدناه كلاً من هاتين الطريقتين.

وتعيد الوظيفة ( )SETKEY القيمة إلى الصفر NIL إذا لم يكن المفتاح مفتاحاً ساخناً أما إذا كان المفتاح ساخناً ، فتعيد هذه الوظيفة كتلة الشيفرة المرتبطة بهذا المفتاح. ويمكنك عندئذ إعادة تعيين كتلة الشيفرة إلى المفتاح عند الانتهاء من استخدامه.

مثال :

```
#include  "setcurs.ch"          // necessary for cursor constants
#include  "inkey.ch"           // necessary for keypress constants


function main
//   statements
myfunc( )
 //   statements
return nil

function myfunc
local oldcursor := setcursor(SC_NONE)      // turn off cursor
local oldexact   := set(_SET _ EXACT , .T. )    // set exact on
```

```
//  note : using SET KEY command to set F1 status below
local oldfl        := setkey(K_F1)
//  this shows how you could structure the SETKEY ( ) code  block
local oldfl0       :=  setkey( K_F10 , { | p , l , v |  whatever( p , l , v ) } )
set key  K_F1 to subhelp
//
// body of function
//
 setcursor( oldcursor )              // restore previous cursor status
 set( _SET_EXACT, oldexact )          // restore previous exact status
 setkey( K-F1 , oldf1 )              // restore previous F1 status
 setkey( K-F10 , oldf10 )            // restore previous F10 status
return nil
```

تشغيل بت الوميض وإيقافه Blink Bit

يجب أن نتحدث عن الوظيفة SETBLINK( ) طالما أننا نتحدث عن موضوع وحدات البرمجة في كليبر. يتساءل كثير من المبرمجين عن كيفية الحصول على ألوان خلفية ساطعة (خاصة بلون أصفر). وإن لدى المبرمج خيارين هما: لوحة أمامية وامضة ، أو خلفية ساطعة ويمكن أن يختار أياً منهما يشاء.

إذا وضع بت الوميض في وضعية الإيقاف ، يمكنك أن تحصل على خلفية ساطعة (على حساب اللوحة الأمامية الوامضة) ، ولاتعتبر هذه خسارة كبيرة. ولا تحتوي إصدارات كليبر السابقة على هذه الوظيفة في حين أن وظيفة كليبر 5.x SETBLINK() تجعلك تقوم بهذا العمل بسهولة متناهية.

إن الوظيفة SETBLINK( ) تم تركيبها على غرار الوظيفة SETCOLOR( ) أي أنه يرجع دائماً إلى الضبط الحالي لبت الوميض ، كما يمكن أن يقبل اختيارياً المتغير المنطقي الذي يضبط حالة بت الوميض. ويراوح المثال التالي بت الوميض بين وضعيتي التشغيل/الإيقاف للحصول على خلفية صفراء:

```
function main
local oldblink := setblink( . f . )
@ @  0 , 0 , maxrow( ) , maxcol( )  box " ******** "  color  ' *n / gr '
```

```
inkey (0)
setblink( oldblink )
return nil
```

وظيفة اختيار اللون COLORSELECT( )

إذا كنت تحتاج بشكل مستمر إلى تغير ضوابط اللون (مثلا : السلسلة الراجعة بالوظيفة (SETCOLOR() لتحديد أحد ضوابط اللون ، فإنك ستحب هذه الوظيفة دون أي شك. فتمكنك الوظيفة ( )COLORSELECT من تنشيط واحد من خمسة تجهيزات للألوان دون تغيير قيمة الوظيفة ( )SETCOLOR. وللتبسيط مثلاً ، يمكنك استخدام ثوابت الإعلان الموجودة في ملف ترويسة COLOR.CH التي يحتوي عليها كليبر ، وهي كما يلي:

manifest constant ثابت البيان	القيمة
CLR_STANDARD	0
CLR_ENHANCED	1
CLR_BORDER	2
CLR_BACKGROUND	3
CLR_UNSELECTED	4

ويبين البرنامج التالي كيفية استخدام الوظيفة ( )COLORSELECT :

```
#include "color.ch"

function main
setcolor ( ' w/r , +w/b , , , +gr/g ' )
? colorselect( CLR_ENHANCED )
? " displays bright white on blue "
? colorselect( CLR_UNSELECTED )
? " displays yellow on green "
? colorselect( CLR_SRANDARD )
? " displays white on red "
return nil
```

التقليل من أمر SELECT

ترى ماهي مساوىء أمر SELECT ؟ فقد تبدو على الظاهر أنها ليست ضارة إطلاقاً. ويستخدمها المبرمج عادة لاختيار منطقة عمل على أخرى غير التي يعمل فيها بحيث تؤثر كافة العمليات المتعلقة بقاعدة البيانات على ملف قاعدة بيانات محدد.

إلا أن أمر SELECT قد يحدث أخطاء خفية في البرامج. وأسوأ ما في هذه الأخطاء الخفية أنها لاتوقف عمل البرنامج ذاته بل بدلاً من ذلك ، فهي تسبب الحصول على نتائج لاتتطابق مع التوقعات التي تتوقع من برنامجك. انظر المثال المبين أدناه:

```
use child index child new
use parent new
do while ! eof( )
    select child
    seek parent ->name
    if found( )
       do while child ->name == parent->name
           delele
           skip
       enddo
    endif
    skip
enddo
```

لقد تأثر كثير من المبرمجين من هذه المشكلة مرة واحدة على الأقل. هل يمكن أن ترى الخطأ في البرنامج؟ فالمشكلة هنا هي أن البرنامج نسي إعادة اختيار "قاعدة بيانات الأب" (parent) بعد حلقة DO WHILE الرئيسة (وليس هذا الأمر واضحاً للجميع كما يبدو). لذلك فإن قاعدة البيانات (الابن) تصبح هي الملف المتحكم بالحلقة Loop وليس هذا بالضرورة ما يريده المبرمج من البرنامج.

ولابد من مراجعة عامل البديل ("<-") قبل التخلص تماماً من العبارة SELECT. ويستخدم المبرمجون عادة هذا العامل للإشارة إلى اسماء ملفات موجودة في منطقة عمل محددة. إلا أن هذا العامل يمكن استخدامه للإشارة إلى أي تعبير في كليبر طالما

أنه سبق بهذا العامل الذي وضع بين قوسين. فماذا أشار العامل إلى منطقة مختارة فإنه سيختار المنطقة المحددة بشكل آلي ويقوم بتنفيذ العملية المطلوبة فيها. ثم يعود لاختيار منطقة العمل السابقة آلياً أيضاً ، وبهذا تستطيع أن ترى أن العامل هذا يجعل العبارة SELECT زائدة أو غير ضرورية.

ملاحظة
ليس هذا التصرف غريباً على كليبر 5.x، إلا أن كثيراً من المبرمجين لا يعلمون عن وجوده).

إن أحد الأفضل الأشياء التي يمكن استخدامها مع العامل ذي الاسم المستعار هي الوظائف الجديدة في كليبر والتي تبدأ أسماؤها بـ: *db. وقد تضمن كليبر مجموعة كاملة من وظائف قواعد المعلومات التي تقابل الأوامر المستخدمة في قواعد البيانات المختلفة كما تبين فيما يلي :

الوظائف التي تبدأ بـ: db	الأمر المطابق
dbAppend( )	APPEND BLANK
dbClearFilter( )	SET FILTER TO
dbClearRel( )	SET RELATION TO
dbCloseArea( )	USE
dbCommitAll( )	COMMIT
dbCreateIndex( )	INDEX ON . . . TO . . .
dbDelete( )	DELETE
dbGoBottom( )	GO BOTTOM
dbGoto(< n >)	GOTO < n >
dbGoTop( )	GO TOP
dbUseArea(. . .)	USE < n > . . .
dbRecall( )	RECALL
dbReindex( )	REINDEX
dbSeek(< exp >)	SEEK < exp >
dbSelectArea(< n >)	SELECT < n >
dbSetIndex([< n >])	SET INDEX TO [< n >]
dbSetFilter( )	SET FILTER TO . . .
dbSetOrder(< n >)	SET ORDER TO < n >
dbSetRelation( )	SET RELATION TO . . .

الجدول مستمر من الصفحة السابقة....

الأمر المطابـــق	الوظائف التي تبدأ بـ: db
SKIP [< n >]	dbSkip([n])
UNLOCK	dbUnlock( )
UNLOCK ALL	dbUnlockAll( )
PACK (use with caution !)	__dbPack( )
ZAP (use with caution !)	__dbZap( )

تحذير

لقد أضيف خيار "التأكد من وجـود أخطـاء" في كليبر 5.2 إلى هـذه الوظـائف المذكـورة. ويجب عدم استخدام هذه الوظائف مالم تكن قد فتحت ملف قاعدة بيانات في منطقة العمل الحالية.

وقد تم توثيق هذه الوظائف جميعها في كليبر ، في ملف دليل نورتون ويمكنك التعرف عليها بشكل أكبر باستخدام الأوامر ذاتها ، وتجميع برنـامجك باستخدام خيـار P/ ، واختبـار النتائج التي يتم تجميعها في ملف PPO. .

لقد تم استخدام الوظائف الجديدة التالية إلى جانب استخدام عوامل الاسم المستعار بحيث يصبح برنامجك أكثر قوة وفاعليـة وتوثيقـاً وذلـك بعـدم الحـاجـة إلى عبـارات اختيـار SELECT علنية. فعل سبيل المثال ، نبين فيا يلي البرنامج السابق عن دليـل الابن/الوالـد دون استخدام عبارة إختر SELECT:

```
use child index child new
use parent new
do while ! parent 0 > ( eof( ) )
   if child->( dbseek( parent->name ) ) // no need to use FOUND( )
      do while child->name == parent->name
         child->( dbdelete( ) )
         child->( dbskip( ) )
      enddo
   endif
   parent->( dbskip( ) )
enddo
```

```
      enddo
   endif
   parent->( dbskip( ) )
enddo
```

ولاشك أن هذا العمل يستدعي مزيداً من الكتابة ، إلا أن الوقت الإضافي القليل الذي تقضيه على أعمال البرمجة سيخفف عنك الكثير من الوقت اللازم لصيانة برامجك لاحقاً. وليس هذا التركيب اللغوي أكثر أمناً وسلامة من أن تتذكر أن تعيد اختيار منطقة العمل الصحيحة بعد استخدام عبارة SELECT ، إلا أنه أيضاً ذاتية التوثيق إذ يمكنك بلمحة سريعة ماهي منطقة العمل المناسبة للعامل المناسب.

لاحظ استخدام الوظيفة DBSEEK( ) في المثال السابق ، فهي تعيد القيمة المنطقية ذاتها كما تفعل وظيفة FUNCTION( ) بحيث يمكنك أن تضغط برنامجك باستخدام الوظيفة DBSEEK( ) بدلاً من استخدام الوظيفة FOUND( ). بل يمكنك أيضاً إرسال متغير منطقي حقيقي (.T.) كمتغير ثالث للوظيفة DBSEEK( ) للقيام بما يسمى SOFTSEEK( ) (دون حاجة لضبط الوظيفة SET SOFTSEEK( ) في وضعية الإيقاف والتشغيل) ولاشك أنها وظيفة محكمة ودقيقة تماماً.

إذا أردت المبالغة في الدقة والجمال لبرامجك فيمكنك أن تضع أكثر من تعبير مستعار ضمن القوسين ، شريطة أن يفصل بينها بفاصلة ، مثل:

```
?  articles - > ( dbskip (1) ,  fieldget (1) )
```



استقلالية وضعية الفيديو

يمكنك كليبر من كتابة برامج يمكن تعديلها آلياً لأية وضعية من وضعيات الفيديو التي يطلبها المستخدم. كما يمكن تسهيل طريقة استخدام كل من طور ٢٥-سطراً أو ٤٣ سطراً أمامك على الشاشة ، أو حتى ٥٠ سطراً من خلال البرنامج الذي تستخدمه. أما الوظائف الثلاثة التي تستخدم لتحقيق كل من هذه الأمور فهي: SETMODE( ) و MAXROW( ) و MAXCOL( ) وإليك بيان كيفية استخدام هذه الوظائف :

وظيفة SETMODE( ) ضبط الوضعية

تمكنك هذه الوظيفة من تغيير وضعية عرض الشاشة ، وتقبل قيمتين رقميتين هما "السطر" و "العمود" <Rows> و <Columns> ، وتحاول الانتقال ما بين هاتين الوضعيتين لاختيار الوضعية المناسبة المطلوبة. ويمكنك تجاوز أي من المتغيرين إذا لم ترغب في تغيير تلك الخاصية (مثلاً : إن الوظيفة SETMODE(50) تغير عدد الأسطر فقط). أما الوظيفة SETMODE( ) فهي تعيد قيمة منطقية إلى حقيقية إذا تم تغيير الوضعية بنجاح ، أو إلى "غير حقيقي" إذا فشلت في تغيير الوضعية المطلوبة. ويعتمد النجاح والفشل في تغيير الوضعية على الأجهزة المستخدمه لديك.

الوظيفة MAXROW( ) / MACXCOL( )

تعيد هذه الوظيفة الوضعية إلى الحد الأقصى من عدد السطور وعدد الأعمدة التي يمكن عرضها على الشاشة. وإن القيم النموذجية لهذه الوضعية هي ٢٤ و ٧٩ إذ أن معظم العمل الذي تقوم به سيكون في وضعية النص القياسية وهي (٢٥×٨٠) إلا أن استخدام الوظيفة SETMODE( ) يسهّل استخدام وضعيات عرض مختلفة في برامجك لذلك

يستحسن استخدام وظيفتي ( )MACXCOL و ( )MAXROW بحيث يمكنك التخطيط طبقاً لذلك. وبدلاً من حفظ شاشة ما واسترجاعها على النحو التالي:

```
oldscrn = savescreen ( 0,  0 ,  24 ,  79 )
restscreen ( 0 ,  0 ,  24 ,  79 ,  oldscrn )
```

ويجب أن تحفظ الشاشة وتسترجعها على النحو التالي:

```
oldscrn = savescreen( 0,  0 ,  maxrow( ) ,  maxcol( ) )
restscreen( 0 ,  0 ,    maxrow( ) ,  maxcol( ) ,  oldscrn )
```

وعند توسيط نص على الشاشة باستخدم الوظيفة ( )MACXCOL+1 كعرض للشاشة بدلاً من 80 ، إذ قد لاتكون دائماً في وضعية ٨٠ عموداً ، فيجب الانتباه.

ويبين المثال التالي كيفية عمل هذه الوظائف الثلاث. وقد تم رسم شاشة عنوان تتضمن مربع في وسطها. ويمكنك بعد ذلك استخدام مفتاح [F1] لتغيير وضعية عرض الفيديو بين وضعيتي التشغيل/الإيقاف ، وسيبقى المربع في وسط الشاشة في كلا الحالتين. ويجب أيضاً ملاحظة التغيير الأقحواني (على شكل زهرة الأقحوان) لوضعية تجهيز الشاشة ما بين ٥٠ سطراً أو ٤٣ سطراً. وقد أعد هذا التغيير خصيصاً بحيث يحاول البرنامج العرض أولاً في وضعية ٥٠ سطراً ، وإذا فشل في ذلك يستخدم وضعية ٤٣ سطراً. ويعتبر هذا الأمر ضرورياً ولازماً لأن معظم مهايئات شاشات في جي أي VGA يمكنها أن تتعامل مع كل من وضعيتي ٤٣ و ٥٠ سطراً.

```
1     #include  "box.ch"
2     #include   "inkey.ch"
3
4     function  main
5     local  oldrows  :  maxrow ( ) ,  oldcols  : =  maxcol ( )
6     videodemo ( )
7     //   reset  video  mode  if  it  was  changed
8     if  maxrow ( )  ! =  oldrows  . or .  maxcol ( )  ! =  oldcols
9        //  chang  color / clear  screen  before  setmode ( ) to avoid  " flash "
10       setcolor ( "w/n ' )
11       scroll ( )
```

```
12      setmode ( oldrows + 1 , oldcols + 1 )
13   endif
14   return nil
15
16   /*
17      Function : videoDemo ( )
18      Purpose : stub function to demonstrate toggling video mode
19   */
20   function videodemo
21   local key
22   titlescreen ( )
23   do while ( key : = inkey ((0) ) ! = K_ESC
24       if key == K_f1
25          togglemode( )
26       endif
27   enddo
28   return nil
29
30   /*
31      Function : ToggleMode( )
32      Purpose : Change video mode and redraw title screen
33   */
34   function togglemode
35   local success
36   if maxrow ( ) 25
37      success : = setmode ( 25 )
38   else
39      success : = ( setmode ( 50 ) . or . setmode ( 43 )
40   endif
41   if success
42      titlescreen ( )
43   endif
44   return nil
45
46   #defin BACK_COLOR  ' +w/b '
47   #defin  INFO_COLOR  ' +w/b '
48   /*
49      Function : TitlesScreen ( )
50   */
51   function titlesScreen
52   local midrow : int ( maxrow ( ) / 2 )
53   local midcol  : int ( maxcol  ( ) / 2 )
54   @ 0 , 0 , maxrow ( ) , maxcol ( ) box repl ( chr ( 197 ) , 9 ) color BACK_COLOR
55   @ midrow - 2 , midcol - 14 , midrow = 2 , midcol = 14 ;
56                      box B_SINGLE = ' ' color INFO_COLOR
57   @ midrow - 1 , midcol - 12 say "Video mode demonstration " ;
58                  color INFO_COLOR
59   @ midrow ,      midcol - 10 say "Now viewing " + ;
60                  ltrim ( str ( maxrow ( ) + 1 ) ) + " lines " color INFO_COLOR
```

```
61   @ midrow + 1, midcol - 12 say "F1 = toggle  ESC = quit "  ;
62                   color INFO_COLOR
63   return nil
```

التحكم بمخرجات الشاشة/الطابعة

لقد واجه المبرمجون مشاكل عديدة أثناء تعاملهم مع الإصدارات السابقة من كليبر أثناء عرضهم لمخرجات العمل. ولعل أكثر المشاكل حدوثاً وإزعاجاً هو الحصول على رسالة SAY...@ التي ترسل إلى الطابعة ، بينما كانت في الأصل موجهة إلى الشاشة. (وذلك لأن تلك الإصدارات اتبعت تجهيز DEVICE بشكل أعمى).

ولحسن الحظ فقد لاحظنا تغييراً جيداً في إصدار كليبر الحالي إذ يمكننا الآن التحكم الدقيق بعدد من وظائف أجهزة الإخراج الجديدة ، وذلك على النحو التالي:

الوظيفة	الوصف
DEUOUT( )	يكتب قيمة لوسيلة الإخراج الحالية
DEVOUTPICT( )	يكتب قيمة لوسيلة مع عبارة صورة
DEVPOS( )	ينقل المؤشر أو رأس الطابعة إلى مكان جديد
DISPOUT( )	يكتب قيمة لوسيلة العرض
MAXCOL( )	يعيد الحد الأعلى للأعمدة بحيث يعرض على الشاشة
MAXROW( )	يعيد الحد الأعلى للأسطر بحيث يعرض على الشاشة
OUTERR( )	يكتب قائمة القيم في وسيلة قياسية للأخطاء
OUTSTD( )	يكتب قائمة القيم في وسيلة قياسية
QOUT( )	يعرض قائمة تعابير في السطر التالي للوسيلة
QQOUT( )	يعرض قائمة تعابير في مكان الوسيلة الحالية
SETPOS( )	ينقل المؤشر إلى مكان جديد على الشاشة
SETPRC( )	يعيد تجهيز مكان رأس الطباعة (بحيث لايكون جديداً وإنما نسبياً)

وظائف تحديد المكان

إن كلاً من وظيفتي DEVPOS( ) و SETPRC( ) تنقل المؤشر و/أو رأس الطباعة.
فتعمل وظيفة DEVPOS( ) على نقل المؤشر أو رأس الطباعة الموجود حسب التجهيز

الحالي للوسيلة DEVICE ، مثـل: المؤشـر إذا كـانت الشاشـة SCREEN ، رأس الطباعة إذا كانت طابعة PRINTER.

وعلى نقيض ذلك ، فإن SETPOS( ) و SETPRC( ) ينطبق على المؤشر أو رأس الطباعة فقط ، بغض النظر عن تجهيز الوسيلة المستخدمة.

وتقبل كل من هذه الوظائف الثلاث متغيرين هما "السطر" و "العمود" <Rows> و <Col> وهي متغيرات رقمية تمثل السطر المراد والعمود المراد الذي يراد وضع المؤشر أو رأس الطباعة عليه. فإذا تم تغيير موقع المؤشر فستختلف قيم كل من "السطر" و "العمود" (وهي الوظائف التي ترجع المؤشر إلى وضعه أو مكانه) ويتم تحديث هذه المواقع طبقاً للتغيير الطارىء. وكذلك فإن تغيير مكان رأس الطباعة سيغير هذه القيم طبقاً للأرقام المطلوبة وسترجع هذه القيم الثلاث دائماً بعد التعديل إلى الصفر NIL تلقائياً.

ويبين الجزء المكتوب أدناه من البرنامج كيفية استخدام الوظائف الثلاث السابقة وكيف تتأثر (أو لاتتأثر) بأمر SET DEVICE:

```
set device to print
setpos( 10, 20 )              // moves cursor
devpos( 1, 2 )                // moves printhead due to DEVICE
set devic to screen
devpos( 8, 0 )                // now moves cursor due to DEVICE
setprc( 10, 10 )              // moves printhead
```

ملاحظات على الوظيفة DEVPOS( )

- إذا طلب من هذه الوظيفة نقل رأس الطباعة إلى سطر أقل من السطر الحالي PROW( ) فستصدر أمراً بإخراج الصفحة قسراً من الطابعة.
- إذا طلب من هذه الوظيفة نقل رأس الطباعة إلى عمود أقل من العمود الحالي PCOL( ) فستصدر أمراً بتقديم السطر وعدد من الفراغات المطلوبة.

■ إذا تم إعادة توجيه الطابعة إلى ملف باستخدام أمر SET PRINTER فتحدث الوظيفة DEVPOS( ) ذاك الملف بدلاً من الطابعة.

وظائف الإخراج

تعالج كل من الوظائف التالية الإخراج الحقيقي الفعلي ، وهي: DEVOUT( ) و DEVOUTPICT( ) و DISPOUT( ) و QOUT( ) و QQOUT( ) و OUTERR( ) و OUTSTD( ) وتقع هذه الوظائف بشكل لطيف ضمن ثلاث مجموعات منطقية لذلك سوف نتعرف عليها من هذا المنطلق.

الوظائف DEVOUT() و DEVOUTPICT() و DISPOUT()

تصدر هذه الخيارات قيمة ، إلا أن كلاً من DEVOUT() و DEVOUTPICT() توجه إخراجاتها إلى الوسيلة الحالية كما يقرره كل من أمر SET DEVICE أو أمر وظيفة SET( ). وتكتب الوظيفة DISPOUT( ) إخراجها دائماً على الشاشة بغض النظر عن الوسيلة الحالية ، ومع أن هذا الأمر خفي ، إلا أنه ذو تأثير كبير كما سنبين فيما يلي.

وتقبل كافة هذه والوظائف متغيراً واحداً هو القيمة التي ستعرض كما أن وظيفة DEVOUTPICT( ) تقبل متغيراً ثانياً والذي هو سلسلة تمثل الصورة PICTURE لعرض القيمة.

وتعيد كافة الوظائف الثلاث القيمة إلى صفر NIL ، ويبين الجزء المبين من هذا البرنامج استخدام هذه الوظائف وعلاقتها بتجهيز الوسيلة الحالية.

```
cstring := " Test message "
devout( cstring )                    // goes to screen
set device to print
devout( cstring )                    // goes to printer
devoutpict( cstring ,  "@!" )      // goes to printer , all upper - case
dispout( cstring )
```

أمر @..SAY

يترجم المعالج الأولي هذا الأمر إلى استدعاءات لكل من DEVOUT( )و DEVPOS( ) على النحو التالي:

```
#command @ <row> , <col> SAY <xpr>                 ;
                              [COLOR <color>]      ;
      => DevPos( <row> , <col> )                   ;;
         DevOut( <xpr> [ , <color> ] )
```

أما وظيفة DEVPOS( ) فتضع المؤشر أو رأس الطابعة في المكان المطلوب ، وتخرج هذه الوظيفة القيمة المطلوبة إلى الشاشة أو إلى الطابعة. (وإذا حددت عبارة PICTURE فسيتم استخدام DEVPICT( ) بدلاً من وظيفة DEVOUT( ).

وتجدر الإشارة إلى أن هذه الوظائف مبنية على حالة الوسيلة الراهنة ، وهذا يعني أن رسالتك قد توجه خطأ إلى الطابعة . ومع هذا ، فمن السهل أن تضمن أن الرسالة @..SAY ستذهب دائماً إلى الشاشة. ويمكنك كتابة أمر يعدّه المستخدم يعتمد على طريقة تجهيز الشاشة باختيار وظيفتي SETPDS( ) و DISPOUT( ) على النحو التالي:

```
#command @ <row> , <col> SSAY <xpr>                 ;
                              [COLOR <color>]       ;
      => SetPos( <row> , <col> )                    ;;
         DispOut( <xpr> [ , <color> ] )
```

ويجب إضافة أمر @..SAY إلى ملف الترويسة لاستخدامه لاحقاً ، كما يجب أن تضع بعين الاعتبار ضرورة أية وظائف تغذية إرجاعية للاستفادة من هذا الأمر. وسيضمن هذا توجيه رسائلك إلى الشاشة دوماً بدلاً من توجيهها خطأ إلى الطابعة.

وظيفتا QQOUT( ) و QOUT( )

تقابل هاتان الوظيفتان استخدام أمر كل من إشارة استفهام واحدة ؟ وإشارتي استفهام ؟؟ وتخرج كل منها قائمة قيم أمامك على الشاشة. والفارق الوحيد بينهما هو أن وظيفة

QOUT() تضع سطراً جديداً قبل القيم ، بينما يلاحظ أن وظيفة QQOUT( ) تخرج القيم المذكورة عند مكان المؤشر فوراً.

وتقبل هذه الوظائف قائمة ذات قيم يفصل بينهما بفواصل ويتم عرضها على الشاشة. ويمكن أن تكون هذه القيم من أى نوع (ما عدا المصفوفات والكتل) ولاداعي لأن تزعج نفسك بتحويل كل شيء إلى سلسلة حرفية. ويبين المثال التالي ماذا نقصد:

```
qout( date( ) ,  5,  "string" ,  .f. )       // perfectly  val
```

ملاحظة

لاحظ أن هذه الوظائف ستضع فراغاً بين كل قيمتين في القائمة.

إن هاتين الوظيفتين تعتمدان على الجهاز (الوسيلة) المستخدم ، ولذلك فهما إما أن يكتبا مخرجاتهما إلى الشاشة مباشرة أو إلى الطابعة. فإذا استخدمت لكتابة النتائج على الشاشة فسيتم تحديث كل من قيمتي "السطر" و "العمود" طبقاً لمكانة المؤشر المحددة. وأما إذا استخدمت لكتابة النتائج على الطابعة فإنه سيتم تحديث كل من PROW( ) , PCOL( ) حسب اللزوم.

قد يبدو لأول وهلة أن هاتين الوظيفتين غير ضروريتين وذلك لوجود كل من أمري ؟ و؟ ؟ (واللتان تترجمان من قبل المعالج الأولي على أنهما استدعاءات لهذه الوظائف وأمثالها) ، إلا أنهما لهما دور خاص وعمل محدد وخاصة في كتل الشيفرة إذا أنك لن تستطيع استخدام أي موجه من موجهات المعالج الأولي **#command** (أو **#xcommand**) ضمن كتلة شيفرة.

وظيفتا OUTSTD( ) و OUTERR( )

تخرج هاتان الوظيفتان قائمة من القيم. وتوجه الوظيفة OUTERR( ) المخرجات إلى وسيلة إخراج خطأ قياسية (stderr) ، بينما تكتب الوظيفة OUTSTD( ) الأخطاء إلى وسيلة إخراج قياسية (stdout) والهدف المفترض لأي منهما هو الشاشة. ويعيد كل من وظيفتي OUTERR( ) , OUTSRD( ) القيمة إلى الصفر NIL ، ويتجاوز إخراج كل من هاتين الوظيفتين شاشة كليبر الأساسية ، وهذا يعني أنه لن تكون لك قدرة على السيطرة على مكانها ، كما أن الوظائف مثل: SETPOS( ) و DEVPOS( ) ليس لهما أي معنى أو قدرة هنا على الإطلاق.

وقد يكون لديك بعض البرامج التي تتطلب عرضاً على كامل حجم الشاشة ، وفي مثل هذه الحالات يمكن أن تستخدم هذه الوظائف لتتجنب تحميل النظام الفرعي لمخرجات الطرفية ، والتي ستوفر عليك بدورها قرابة ٢٥ كيلو بايت في الملف التنفيذي للبرنامج. ويمكن تسهيل هذه العملية باستخدام الموجه #inelude في ملف الترويسة SIMPLE10.CH الموجود في كليبر ، إذ سيعيد هذا الملف إعادة تعريف كل من أمري ؟ و ؟ ؟ إلى جانب الأمر الذكي جداً وهو ACCEPT والذي لا يستخدم النظام الفرعي لمخرجات الطرفية.

وعلى خلاف وظائف إخراج كليبر الأخرى فيمكن أن تستخدم إعادة توجيه " دوس " باستخدام OUTSTD( ) ، إلا أن وظيفة OUTERR( ) ستتجاوز إعادة التوجيه أيضاً. ويستخدم البرنامج التالي كثيراً من وظائف الإخراج ويبين أين ستوجه:

```
function main
set device to printer
qout ( "* to screen " )
dispout ( "* to screen " )
devout ( "* to printer " )
outstd ( "* redirection " )
outerr ( "* to screen " )
return nil
```

أمثلة عن الإخراج

توسيط النص

يعتبر هذا الأمر حاجة أساسية تستخدم في كل برامج كليبر ، ونبين فيما يلي وظيفتين معرفتيَن من قبل المستخدم وتوسط الوظيفة ( )CENTER النص إما على الشاشة أو على الطابعة ، وذلك طبقاً للوسيلة المستخدمة. وأما الوظيفة ( )SCRNCENTER فتوسط النص على الشاشة دائماً ، وذلك باستخدام كل من وظيفتي ( )DISPOUT و ( )SETPOS ، على النحو التالي:

```
#xtranslate  CENTER ( <row> , <msg>  [ , <width>  ] )  =>   ;
   DevPos(  <row> ,  int ( (  IF ( len ( #<width> )  =  0,     ;
                maxcol ( ) +1, val ( #<width> ) )  - len ( <msg> ) )  / 2 ) )  ;   ;
   DevOut (  <msg>  )

#xtranslate  SCRNCENTER ( <row> , <msg>  [ , <width>  ] )  =>   ;
   SetPos (  <row> ,  int ( (  maxcol ( )+1  -  len ( <msg> ) )  / 2 ) )  ; ;
   DispOut (  <msg>  )

function test
set device to print
center ( 1,  "this goes to the printer " ,  136 )
scencenter ( 1,  " this goes to the screen " )
return nil
```

ولاحظ ، كما ذكرنا سابقاً ، أن كلا من وظيفتي ()SCRNCENTER و ()CENTER يستخدما الوظيفة ( )MAXCOL لتحديد عرض توسيط رسالتك. ويبدو جلياً من هذا أنه سيتم تغيير وضعية العرض بحيث توفر كثيراً من الوقت اللازم لتسجيل برامجك إذا قررت تغيير واجه المستخدم للرسوم في جهازك مستقبلاً.

عرض أرقام الصفحات أثناء الطباعة

يبين البرنـامج التـالي كيـف يمكنـك استخدام وظيفـتي DISPOUT( ) و SETPOS() لعرض أرقام الصفحات أثناء طباعة تقرير ما. وإن اسـتخدام هـذه الوظـائف يستثنيك مـن الانتقال إلى تجهيز الوسيلة DEVICE جيئة وذهاباً (والذي يمكن أن يسبب لك اضطراباً بسرعة كبيرة).

```
#include "box.ch"

function report
use customer
2 11, 28 , 13 , 52 box B_SINGLE + chr ( 32 )
2 12 , 30 say " Now printing page "
set device to print
heading ( )
do while ! eof ( )
   // code to print report
   if prow ( ) > 57
      heading ( )
   endif
   skip
enddo
eject
set device to screen
return nil

function heading
static page : = 1
2 0 , 0 say "Customer List - Page " + 1 trim ( str ( page ) )
settpos ( str ( page + + , 3 ) )        // increment page counter
return nil
```

تعلن وظيفة heading( ) الصفحة على أنها متغير سـاكن ، وهـذا يعـني أنهـا ستحتفظ بقيمتها في كل مرة تدخل فيها هذه الوظيفة. وتنقل وظيفة heading( ) رأس الطابعة إلى رأس الصفحة ويعرض ترويسة مختصرة مع رقم الصفحة. ثم يعرض بعد ذلك رقم الصفحة على الشاشة في مربع الرسالة ، ويزيد العداد لزيادة عدد الصفحات.

ويبين المثال التالي هذا المبدأ الأولي بتوسع. فهو يعرض أولاً السجلات الحالية والإجمالية التي تمت طباعتها ، ثم يمكنك من التوقف اللحظي ، أو إنهاء العمل والخروج أو البدء بالتقرير

من جديد (وهذا أمر رائع إذا حدث عطل في الطابعة). ويجب استدعاء كل من الوظائف التالية: StopPrint( ) و Printing( ) و StartPrint( ) لاستخدامها في برامجك ، كما هو موضح في صيغة البرنامج التالي:

```
startprint ( recno ( ) )
do while condition . and . printing ( )
   // code to print data
   skip
enddo
stopprint ( )
```

لاحظ أن وظيفة StartPrint( ) تقبل متغيراً واحداً ، وهذا هو رقم السجل الأول لتقريرك وسيتم استخدام هذا الرقم من جديد إذا أردت إعادة الطابعة من جديد. كما يرجى ملاحظة استدعاء الوظيفة Printing( ) على الخط ذاته ، كجزء من العبارة الشرطية DO WHILE إذ يضمن هذا أنه سيستدعى في كل مرة تتم فيها معالجة سجل ما حسب المثال التالي:

```
 1  #define TEST                 // identifier to compile test program
 2
 3  //   preprocessor directines
 4
 5  #define TOP                  scrnbuff[1]
 6  #define LEFT                 scrnbuff[2]
 7  #define BOTTON               scrnbuff[3]
 8  #define RIGHT                scrnbuff[4]
 9  #define CONTENTS             scrnbuff[5]
10  #define BOXCOLOR             "+W/B"      // color for message box
11  #define MSGCOLOR             "+GR/B"     // color for status message
12
13  //   screen - specific @ . . SAY
14  #xcommand @ <row>, <col>, SSAY <xpr> [COLOR> ] ;
15            => setPos ( <row>, <col> ) ; DispOut ( <xpr> [ , <color> ] )
16
17  //   screen - specific Center ( )
18  #xtranslate SCRNCENTER ( <row>, <msg> )  =>  ;
19              SetPos ( <row>, int ( ( maxcol ( ) +1 - len ( <msg> ) )  / 2 ) )  ; ;
20                DispOut ( <msg> )
21
```

```
22 #include "box.ch )
23 #include "inkey.ch"
24
25 //  file-wide static variables
26
27 //  housekeeping thing - - these are declared file-wide because
28 //  one function saves them and another restores them
29 static scrnbuff  : = { 11, 18, 14, 61 }
30 static oldcursor                  // previous cursr state
31 static oldcolor                   // previous color
32
33 static firstrec                   // in the event of a restart
34 static counter                    // counts records processed
35 static page  : = 1                // self-explanatory I hope !
36
37
38 #ifdef TEST
39
40 /*
41     REPORT( ) - - stub program for testing these functions
42                      pass it the name of a database
43 */
44 function report (dbf_name)
45 if dbf_name != NIL
46    use ( dbf_name )
47    startprint( recno ( ) )
48    do while ! eof( ) . and . printing( )
49        // display first two fields in . DBF
50        @ prow( ) +1 , 0 say fieldget(1)
51        devout( fieldget(2)
52        skip
53    enddo
54   stopprint( )
55 endif
56 return nil
57
58 #endif
59
60
61 /*
62    Startprint( <startrec> )
63    Initialize counter , display message box , turn printer on
64    Parameter : <startrec> = = starting record numbere ( used in the
65                    event of a restart )
66    Returns :    Nothing of consequence
67    NOTE : must be called prior to calling printing ( )
68 */
69 function startprint ( recno )
70 //  initialize FIELD-wide statics
```

```
71 firstrec       := recno                          // restart
72 oldcursor      := setcursor (0)                  // shut off cursor
73 counter        := 0                              // reset record counter
74 aadd ( scrnbuff , savescreen ( TOP, LEFT , BOTTOM , RIGHT ) )
75 oldcolor       := setcolor ( BOXCOLOR )
76 @ TOP , LEFT , BOTTOM , RIGHT box B_SINGLE + chr (32)
77 @ TOP + 1 , LEFT + 2 ssay " Now printing record "
78 SCRNCENTER ( TOP + 2 , " Pause  Quit  Restart " )
79 set device to print
80 setprc ( 58 , 0 )                    // force initial page eject
81 return nil
82
83
84 /*
85    printing( )
86    Initialize counter , display message box , turn printer on
87    Parameter : Nada
88    Returns :    Logical value : True -- continue printing
89                                 False -- printing aborted
90 */
91 function printing( )
92 local key := inkey( )
93 local buffer
94 local ret_val := .t .
95
96 //  inspect last keypress
97 do case
98
99   /* p -- pause */
100  case key == 80 . or . key == 112
101     buffer := showmsg ( "Paused . . . press any key to continue " )
102     inkey (0)
103     restscreen ( TOP + 2 , LEFT + 1 , BOTTOM - 1 , RIGHT - 1 , buffer )
104
105   /* Q ( or Esc ) -- Quit */
106   case key == 81 . or . key == 113 . or . key == K_ESC
107      buffer := showmsg ( "press Q to confirm quit " )
108      if ( key := inkey (0) ) == 81 . or . key == 113
109         ret_val := . f .
110      else
111         restscreen ( TOP + 2, LEFT+1, BOTTOM-1, RIGHT-1, buffer )
112      endif
113
114   /* R -- Restart */
115   Case Key == 82  . or . Key == 114
116        buffer := showmsg ( " press to confirm restart " )
117        if ( key := inkey ( 0 )   == 82 . 82 . key == 114
118            page := 1
119            go firstrec
```

```
120          counter  := 0                          // reset  record  counter
121              setprc ( 58 , 0 )
122          endif
123          restscreen ( top  + 2,  LEFT +  1,  BOTTOM - 1,  RIGHT -  1,  buffer )
124
125    endcase
126    if  Prow ( )  57
127      @ 0,  0  say  " page " + 1trim (str ( page + + ) )  // increment  page  #
128     endif
129     setpos ( TOP  + 1,  LEFT + 22 )
130     dispout  ( Padr ( 1 trim ( str ( + +counter ) )  +  " of "  +  ;
131                 1trim ( str ( 1astrec ( ) ,     20 ) )
132      return  ret_val
133
134      / *
135           Stop print ( )
136           C1osing  page  eject ,  restore  screen ,  turn  off  printer
137            parameter :   Nada
138            Returns :     Nothing worth  writing  home  about
139      * /
140      function  stopprint ( )
141      eject
142      set   device  to  screen
143      restscreen ( TOP ,  LEFT, BOTTOM, RIGHT,  CONTENTS )
144      setcolor  ( oldcolor )
145      setcursor ( oldcursor )
146      return  nil
147
148
149      / *
150           static  function  showMsg ( )
151           Used by printing ( )  to  display  message  for  pause / Quit / Restart
152           parameter: Message to display
153           Returns:    Affected  portion  of  screen  for  later  restoration
154      * /
155      static  function  showmsg ( msg )
156      LocaL buffer  : = savescreen ( TOP  + 2, LEFT+1,  BOTTOM -1, RIGHT _  1 )
157     scroll ( TOP + 2,  LEFT +  1,  BOTTOM - 1, RIGHT  -  1,    0 )
158     setcolor  ( MSGCOLOR)
159     SCRNCENTER ( TOP + 2,  msg )
160     setcolor  ( BOXCOLOR )
161     return  buffer
```

الذاكرة المؤقتة لمخرجات الشاشة

إن الشاشات التخيلية هي الشاشات التي لا تظهر على الشاشة الحقيقية. ويتم إعداد هذه الشاشات بشكل عام في الذاكرة ثم تظهر أمامك على الشاشة الحقيقية عند اللزوم. وتبين لنا وظيفتان من وظائف كليبر مبادىء نظام النوافذ التخيلية وهما: DISBPGIN( ) و DISPEND( ). وسنبين كلا منهما فيما يلي:

إن الوظيفة DISPBGIN( ) تعيد توجيه كافة مخرجات كليبر من الشاشة الحقيقية إلى الشاشة التخيلية. ويتضمن هذا الإخراج القادم من خلال كل من الأوامر التالية : @..SAY و @..BOX و QOUT( ) و QQOUT( ) و DEVOUT( ) و DISPOUT( ).

وأما الوظيفة DISPEND( ) فهي شبيهة من حيث المفهوم بالوظيفة RESTSCREEN( ) ، إلا أنه بدلاً من استرجاع الشاشة التي تم حفظها سابقاً ، فهي تعرف أمامك على الشاشة محتويات الشاشة التخيلية.

ونبين فيما يلي مثالاً بسيطاً عن هاتين الوظيفتين في البرنامج التالي ، ومع أنه تم رسم إطار فلن يمكن مشاهدته إلا بعد أن تضغط على مفتاح ما ، كما هو في المثال التالي:

```
function main
dispbegin( )
@ 0, 0, maxrow( ) , maxcol( ) box replicate ( " * " , 9 ) COLOR '+w / r'
inkey ( 0 )
dispend ( )
return nil
```

أمثلة على كل من وظيفتي DISPBEGIN( ) و DISPEND( )

يحتوي البرنامج التالي على مثالين عن الشاشات التخيلية . وتبين الوظيفة Virtual( ) أنه ربما لايظهر شيء أمامك على الشاشة الحقيقية فإن الوظيفة SAVESCREEN() ستقوم

على الأقل بحفظ محتويات الشاشة التخيلية في متغير محدد لتتمكن من استرجاعها فيما بعد. وهذا يعني أن بإمكانك إعداد أي عدد من الشاشات بحيث تظهر على الشاشة في مختلف الأوقات أثناء التعامل مع البرنامج.

إن وظيفة ( )Virtual (التخيلي١) تستدعي الوظيفة ( )DISPBEGIN لتعيد توجيه الإخراج إلى الشاشة التخيلية. وهنا يتم رسم ثلاث إطارات شاشات تبادلية زرقاء وحمراء وبنفسجية. ويتم حفظ كل شاشة من هذه الشاشات باستخدام أمر SAVESCREEN( ) وتتم إضافتها إلى مصفوفة الشاشات. ثم يتم استدعاء وظيفة DISPEND( ) والتي تضع محتويات الشاشة التخيلية الثانية على الشاشة الحقيقية. ثم تضع الحلقة DO WHILE والتي تبادل بين الشاشات المحفوظة الثلاث كل نصف ثانية. اضغط على أي مفتاح بعد أن تقتنع بما ترى.

أما المثال الثاني فيقدم منطقاً لاسترجاع حجم الإطار باستخدام مفاتيح الاتجاهات وهذه عملية ذات مرحلتين ، الأولى: هي أن تسحب الزاوية العليا اليسرى للإطار ، ثم تسحب الزاوية اليمنى السفلى منه. ولاشك أنك تعرف كيفية عمل كل من أسهم الاتجاهات الأربعة المختلفة : إلى أعلى [↑] ، وإلى أسفل [↓] ، وإلى اليمين [→] وإلى اليسار [←].

أما الوظيفة Virtual2() (التخيلي ٢) فتقبل متغيراً واحداً فقط وهو <noflicker> وهو قيمة منطقية ، فإذا مررت القيمة المنطقية (.T.) فستستخدم الوظيفة Virtual2() الوظيفة DISPEND( ) ووظيفة DISPBEGIN( ) لحذف إهتزاز الشاشة لدى استرجاع الشاشة الموجودة تحت الإطار. أما إذا مررت القيمة المنطقية "غير حقيقي" (.F.) فسترى اهتزازاً مزعجاً ، وخاصة إذا ضغط على زر الاتجاه إلى اليسار أو اليمين وأبقيت أصبعك ضاغطاً عليه. جرب هذه العملية وتأكد من الإزعاج بنفسك.

وتستدعي الوظيفة Virtual2() مرتين من قبل سرير الاختبار ، مرة مع الاهتزاز ، ومرة أخرى دونه. ولم يكن بالإمكان حذف الاهتزاز المزعج في الإصدارات السابقة من كليبر Summer'87 .

أما الاستخدامات الأخرى لهذه الوظائف فهي تتضمن شاشات إدخال البيانات التي تتضمن كثيراً من أوامر SAY..@ و GET..@ . وإذا سبقت شاشة إدخال البيانات بوظيفة DISBEGIN( ) وأتبعتها بوظيفة DISPEND( ) (قبل أمر READ مباشرة) فستكون الشاشة واضحة جداً. انظر المثال التالي:

```
1   # include  "box . ch"
2   # include  "inkey . ch "
3
4   #define  INFO_COLOR   'n / bg'
5
6   function  main
7   virtua11 ( )
8   virtua12 ( . f . )
9   virtua12 ( . t . )
10  return  nil
11
12   / *
13        function :  virtua11 ( )
14        purpose :   show  that  SAVESCREEN ( )   works  with  DISPBEGIN ( )
15   * /
16    function  virtuall
17    local  x  : =  1
18    local  screens { 3}
19    dispbegin ( )
20    setcursor ( 0 )
21    dispbox ( 0,    0,   maxrow ( ) ,    maxcol ( ) ,   replicate ('1' ,  9 ) ,  'w/b' )
22    dispbox ( 6,     10,   maxrow ( ) - 6,   maxcol ( ) - 10,  replicate ('2' , 9 ), 'w/ r' )
23    dispbox ( 10,  20, maxrow ( ) - 10 , maxcol ( ) - 20, replicate( '3' ,  9 )  'w/ rb' )
24    screens { 1 } : =    savescreen  ( 0 ,     0 ,   maxrow ( )    maxcol ( ) )
25    dispbox( 0 ,   0,   maxrow ( ),   maxcol ( ) , replicate ('2' ,     9) ,    'w/r'  )
26    dispbox( 6 ,  10 , maxrow ( ), 6 , maxcol ( ) - 10 , replicate ( '3'  ,  9 ) , 'w/ rb' )
27    dispbox( 10, 20 , maxrow ( ) - 10 , maxcol ( ) - 20 , replicate ( '1' , 9 ) , 'w/ b' )
28    screens { 2 }   : =   savescreen  ( 0,       0 ,    maxrow ( ) ,    maxcol ( ) )
29    dispbox( 0 ,  0 , maxrow ( ) , maxcol ( ) ,    replicate  ( ' 3 ' ,   9 ) ,   ' w/ rb ' )
30    dispbox( 6 , 10, maxrow ( ) -6, maxcol ( )-10, replicate( ' 1 ' ,      9 ) ,  ' w/ b ' )
31     dispbox( 10,  20, maxrow( )-10,  maxcol( )- 20, replicate( ' 2 '  , 9 ) ,  ' w/ r' )
32     screens { 3 }   : =   savescr een  ( 0 ,    0 ,   maxrow ( ),  maxcol ( ) )
33    scro11 ( )          / /    so   that   we   start     out   with  a  blank  screen
34    dispend ( )
35    do  while    inkey ( . 5 )    = =  0
36        if   x   <  3
37           x + +
38        else
39           x   : =   1
```

```
40        endif
41        restscreen ( 0 ,  0 , maxrow ( ) , maxcol ( ) , screens { x } )
42     enddo
43     return nil
44
45
46    /*
47          Function : Virtual2 ( )
48          Purpose :  Resize a box without screen flicker
49    */
50    function Virtual2 (noflicker )
51    local  t : = maxrow ( ) /2  -  2
52    local  1  : =  10
53    local  b  : = maxror ( ) / 2 + 2
54    local  r  : =  moxcol ( )   -  10
55    local   x
56    local  oldscrl
57    local  key
58    local  boxstring
59    setcursor (0)
60
61    // drew bogus backdrop to prove the point
62    for x  : = 0  to  maxrow ( )
63        @  x , 0  say replicate (chr(x) , maxcol ( )  +  1 ) color 'w/b'
64    next
65
66    oldscen  : = savrescreen ( 0 , 0 , maxrow ( ) , maxcol ( )  )
67    boxstring  : = chr (4)  + substr (B_SINGLE , 2 )  +
68    @ t , 1 , b , r box boxstring color INFO_COLOR
69    @ t + 1,  1 + 2  say "Press arrow keys to resize" color INFO_COLOR
70    @ t + 2,  1 + 2  say "screen flicker"  + ;
71              if ( noflicker , "dis" , "en" )  +  "abled" color INFO_COLOR
72    inkey (2)
73
74    //  first anchor the top left corner
75    do while key ! = K_ESC  . and . key ! = K_ENTER
76       key  : = inkey (0)
77      do case
78         case key = = K_LEFT .and . 1 > 0
79            1 - -
80         case key = = K_RIGHT . and . 1< r + 1
81            1 + +
82         case key = = K_UP  . and . t > 0
83            t - -
84         case key = = K_DOWN . and . t < b - 1
85           t + +
86    endcase
87    if noflicker
88          dispbegin ( )
```

```
89    endif
90    restscreen ( 0 , 0 , maxrow ( ) , maxcol ( ) , oldscrn )
91    @ t , 1 , b , r box boxstring color INFO_COLOR
92    if noflicker
93       dispend ( )
94    endif
95  enddo
96
97  key : = 0
98  boxstring :=substr( B_SINGLE, 1, 4)+ chr(4)+substr(B_SINGLE, 6 )+ ' '
99
100
101 // now anchor the bottom right corner
102 do while key ! = K_ESC . and key ! = K_ENTER
103     key : = inkey (0)
104     do case
105    case key = = K_LEFT .and . r > 1 + 1
106            r - -
107           case key = = K_RIGHT . and . r < maxcol
108             r + +
109           case key = = K_UP . and . b > t + 1
110             b - -
111           case key = = K_DOWN . and . b <  maxrow ( )
112             b + +
113   endcase
114   if noflicker
115      dispbegin
116   endif
117   restscreen ( 0 , 0 , maxrow ( ) , maxcol ( ) , oldscrn )
118   @ t , 1 , b , r box boxstring color INFO_COLOR
119   if noflicker
120      dispend ( )
121   endif
122 enddo
123 return
```

ملاحظة لمستخدمي كليبر 5.2

يمكنكم الآن نسخ عدة استدعاءات لوظيفة ( )DISBEGIN داخل بعضها (تعشيش). ولن يتم تجديد الشاشة إلا بعد أن يتم إصدار الرقم المطابق لوظيفة ( )DISPEND فيها. وإذا أردت معرفة عدد المرات التي تم فيها استدعاء الوظيفة ()DISPBEGIN ، فيجب استخدام الوظيفة ( )DISPCOUNT كما هو موضح في المثال التالي:

```
function main
dispbegin ( )                    // DISPCOUNT ( ) is set to 1
scroll ( )
@ 0 , 0 say "first massage" color "w/r"
dispbegin ( )                    // DISPCOUNT ( ) is set to 2
@ 1 , 0 say "second message" color "w/b"
dispend ( )    // DISPCOUNT ( ) is set to 0 , screen is not
displayed
inkey (0)
dispend ( )          // DISPCOUNT ( ) is set to 1 , screen is
redisplayed
return nil
```

تحذير

كما يجب أن تلفت الانتباه إلى عدم استخدام كل من الوظيفتين ( )OUTERR و ( )OUTSTD ، ضمن وظيفتي ( )DISPEND أو كتلة ( )DISPBEGIN ، علماً بأنهما يستخدمان استخداماً خاصاً. وإنك إذا فعلت ذلك خطأ فستقذف محتويات العبارة الناتجة عن استخدام هاتين الوظيفتين أمامك على الشاشة الحقيقية ، وبهذا فإنك لن تحصل على الهدف المطلوب من استخدامهما بشكل صحيح.



المصفوفات ARRAYS

إن أحد أهم مزايا كليبر أنه قادر على معالجة كل من المصفوفات القاسية والمرنة على حد سواء. وسيسر المبرمج جداً بمعرفة إمكانية كليبر على تحجيم المصفوفات وتداخلها فيما بينها. أما إذا لم تستخدم المصفوفات من قبل ، فسنبين لك في المثال التالي كيفية استخدام هذا الأمر.

ماهي المصفوفة ؟

المصفوفة هي نوع من أنواع بيانات كليبر يحتوي على مجموع من قطع بيانات أخرى. وتحفظ هذه الأجزاء من المعلومات في "عناصر". وقد يحتوي "المجموع" على حد أعلى قدره ٤٠٩٦ من هذه العناصر. وعلى خلاف بقية لغات البرمجة ، فإن كليبر يمكنك من مزج مختلف أنواع البيانات في أية مصفوفة من المصفوفات. وهذه الميزة قوية جداً بحيث تسمح لك تمثيل تركيبات بيانات لغة سي C (إلى جانب أمور كثيرة أخرى).

ولعل جمال استخدام المصفوفات بدلاً من المتغيرات العديدة هو أن عناصر المصفوفة تجمع منطقياً إلى جانب بعضها ، بحيث يسهل معالجتها جميعاً بدلاً من معالجة كل منها على حدة . وعند إنشاء مصفوفة ما ، فإنك تحدد مرجعها إلى اسم متغير مثل: local aDay[7].

ويرجى الانتباه إلى أن عبارة aDay لاتحتوي بذاتها على مصفوفة ، بل إنها تحتوي على مرجع للمصفوفة. وهذا التمييز أساسي وحيوي جداً إذ أن مفهوم تداخل المصفوفات يعتمد اعتماداً كلياً عليه.

وبعد إنشاء مصفوفة وتحديد مرجعها إلى متغير ما ، فلا بد من الإشارة إلى عناصر بياناتها. وتستخدم "الرموز السفلية" لتنفيذ هذا الأمر. وتبين الرموز السفلية باستخدام القوسين

المعقوفين [ ]. فمثلاً ، أن السطر التالي من البرنامج يشير إلى العنصر الخامس من المصفوفة aDay.

```
? aDay [ 5 ]
```

يمكن سلسلة الرموز السفلية على شكل زهرة الأقحوان للإشارة إلى عناصر داخل مصفوفات متداخلة. وسنبين أمثلة على هذا النوع لاحقاً في هذا الفصل. كما سنرى أيضاً أن مجموعة الرموز السفلية قد تتبع إشارة إلى مصفوفة (مثلاً: استدعاء وظيفة ، إذا رجعت هذه الوظيفة إلى مصفوفة).

ملاحظة

إذا كنت تبرمج بلغات أخرى مثل لغة C (على سبيل المثال) فيجب الانتباه إلى أن مصفوفات كليبر هي من نوع "الاعتماد على أول حرف" (one-based) ، أي أن العنصر الأول في مصفوفة كليبر يشار اليه على أنه المصفوفة[1] بدلاً من المصفوفة صفر [0].

إعلان المصفوفات وتأسيسها

يتيح لك كليبر ثلاث طرق لإنشاء المصفوفات وهي:

- طريقة المصفوفة الثابتة "Fixed": يتيح لك كليبر استخدام الإعلانات الجالية (وبهذا تنشأ) مصفوفة ذات طول عشوائي.

```
local days [ 7 ]
static  totals [ 50 ]
```

- المصفوفة العادية ARRAY( ) : تعمل هذه الوظيفة بالطريقة ذاتها التي تعمل بها طريقة المصفوفة الثابتة في كليبر ، وستبين الأمثلة التالية أنه يمكن تحقيق النتائج ذاتها التي تم تحقيقها في المثال السابق : والاستثناء الوحيد لهذا هو أننا لن نستطيع تأسيس مصفوفة ساكنة STATIC باستخدام وظيفة.

```
local days  := array(7)
```

وقد تتساءل هنا لماذا يجب أن تستخدم وظيفة "المصفوفة" ARRAY() بدلاً من "الإعلان" للعادي المعروف ؟ والجواب على هذا هو أنه يمكن استخدام المصفوفة ARRAY( ) لإنشاء مصفوفات أخرى داخل التعابير أو كتل الشيفرة. وإليك مثالاً عن المكان الذي يفضل فيه استخدام وظيفة "المصفوفة":

```
aadd( myarray ,  array ( 10 ) )
```

إذ أن هذا يضيف مصفوفة ذات عشرة عناصر إلى المصفوفة المسماة MYARRAY. وليس هناك أية طريقة يمكن أن نقوم بها بمثل هذا العمل. ولن يستجيب المجمِّع إذا كتبت أمراً كالأمر التالي في برنامجك:

```
aadd( myarray ,  local  array ( 10 ) )
```

- المصفوفة الحرفية Literal : يمكنك كليبر 5.x من إعلان مصفوفة وتأسيسها بانتزاعها من مكان واحد. فلنفترض أنك تريد إعلان مصفوفة وملأها ببعض الأسماء فيمكنك تمثيل هذه المصفوفة حرفياً كما يلي:

```
local  names  :=  {  "Joe" ,  "Paul;" ,  "carol"  ,    ;
                        "Justin" ,  "Jennifer" ,  "Mary"  }
```

- وتعتبر الأقواس الجعدة "{ }" تركيباً لغوياً لازماً للمصفوفة الحرفية. ويحدد القوس "}" بداية المصفوفة الحرفية ، بينما يحدد القوس التالي "{ " نهايتها. ويعتبر كل ما في داخل هذين القوسين محدداً بشكل متتال لعناصر المصفوفة. ولن تعود بحاجة لكتابة عدد العناصر التي تريد أن تضعها في المصفوفة بل تستطيع المصفوفة تحديد العدد بذاتها.

كما يمكنك أيضاً ترك المسافة بين القوسين الجعديين فارغة ، وفي هذه الحالة سينشيء كليبر مصفوفة. أما لماذا تحتاج إلى المصفوفة الفارغة ؟؟. فيمكنك تغيير حجم المصفوفات بشكل ديناميكي وفعال باستخدام هذه الميزة .

تحذير

يجب الانتباه التام إلى ضرورة عدم استخدام الأقواس العادية المستقيمة بدلاً من الأقواس الجعدة عند إعلان مصفوفات حرفية ، علماً بأنها قد تبدو لك طبيعية وعادية للاستخدام وخاصة لأننا كمبرمجين لا نستخدمها للإشارة إلى عناصر المصفوفات. فمثلاً ، قد تظن أنك تعلن عن مصفوفة :

```
local myarray := [ ]
```

والحقيقة أنك بهذه الطريقة تعلن عن سلسلة حرفية وليست مصفوفة. كما أن أية شارة لاحقاً إلى هذا المحتوى سيتسبب في أخطاء تقع أثناء التشغيل وقد تحصل على كلمات لا معنى لها.

تأسيس عناصر المصفوفة

هناك ثلاث طرق لتأسيس عناصر مصفوفة array فردية ، ويمكن استخدام عامل التعيين الموثوق ، وهو أمر STORE ، أو طريقة التمثيل الحرفية. وسنبين فيما يلي أمثلة على هذه الطرق:

```
local myarray := { "Clipper" , "5" , "Oh!" }
marray[40] := 75
store 0 to marray[1] , marray[3]
```

كما أن هناك طريقة أخرى لتأسيس عناصر مصفوفة ، وهي أن تدع كليبر يؤسسها لك!!.
فعند إعلان مصفوفة ذات طول محدد فسيتم تأسيس عناصر المصفوفة جميعها بشكل آلي إلى القيمة "صفر" NIL.

وإذا أردت تأسيس مصفوفة كاملة إلى الرقم ذاته ، يمكنك استخدام الوظيفة AFILL( ) ، على النحو التالي:

```
local myarray[100]
? myarray[8]            //NIL
afill(myarray, 5 )
? myarray[8]            // 5
? myarray[10]           // 5
```

ويمكن استخدام الوظيفة AFILL( ) لملء عناصر محددة في المصفوفة ، وسترى في العبارة التالية أنه سيتم ملؤ المصفوفة بالرقم (5) بدءاً من العنصر (10).

```
afill(myarray, 5, 10 )
```

كما يمكن أيضاً تحديد رقم العنصر الذي يراد ملؤه بالذات ، سيتم ملؤ المصفوفة التالية بالرقم (5) ولعشرين عنصراً بدءاً من العنصر (10) :

```
afill(myarray, 5, 10, 20 )
? myarray[29]     // 5
? myarray[30]     // NIL
```

تنبيه

إن استخدام الوظيفة afill( ) لملء مصفوفة ما بمصفوفات متداخلة سيكون خطراً جداً. وسنبين هذا في موضعه من هذا الكتاب.

الإشارات المتعددة إلى مصفوفة واحدة

ذكرنا سابقاً أنه عند إنشاء مصفوفة ما فإننا نعين إشارتها (أو مرجعها) إلى اسم متغير. فإذا نقلت هذا المتغير إلى متغير آخر سيصبح لديك متغيران يشيران إلى المصفوفة ذاتها. ويبين جزء البرنامج التالي هذا الأمر بتعيين محتويات المصفوفة aNother.

```
local  aDay[7]
local  aNother  :=  aDay
aNother[4]   :=   "TEST"
?  aDay[4]  //     "TEST"
```

فعند تعيين العنصر الرابع مـن المصفوفـة المشـار إليهـا باسـم aNother ، فإننـا بالتـالي نعـين العنصر ذاته إلى مصفوفة aDay أيضاً. وهذا أمر هام جداً يجب الانتبـاه اليـه أثنـاء التعـامل مع كليبر.

فإذا أردت أن تضمن أنـك تنشـيء إشـارات إلى مصفوفـات مستقلة مختلفـة يجـب استخدام وظيفة كليبر المسماة ACLONE( ) التي سنتحدث عنها بعد قليل.

مساواة المصفوفة

يمكن مقارنة إشارات مصفوفـة مـا بأخرى باستخدام عـامل المسـاواة ("=") أو المسـاواة التامة ("==") ، ولدى القيـام بذلـك سـيرجع كليـبر العـامل المنطقـي الحقيقـي (.T.) إذا كانت إشارات كل من المصفوفتين تشير إلى عنـوان الذاكـرة ذاتـه. وإنـه لـن يقـوم بعمـل حلقة خلال كل مصفوفة ليقارن محتويات كل عنصر. ويبين المثال التالي هذا العمل:

```
local array1  :=  { 1, 2, 3 }
local array2  :=  array1
local array3  :=  {1, 2,  3 }
?  array1   ==  array2       //  .T.
?  array1   ==  array3       //  .F.
```

- بما أن المصفوفة ARRAY2 قد عين لها قيمـة المصفوفـة ARRAY1 فإنهمـا يحتويـان عنوان الذاكرة ذاته ، ولذلك فهما يعتبران متساويين.
- ومع أن المصفوفة ARRAY3 تحتوي على ثلاثة عناصر كل منها له القيمة ذاتها مثـل تلـك الموجـودة في المصفوفـة ARRAY1 ، فهمـا تمثـلان مصفوفـتين مسـتقلتين عـن بعضهما تماماً ، فلا يعتبرهما كليبر متساويتين.

اختبار نوع المصفوفة وطولها

تعمل هذه الوظيفة LEN( ) بطريقتين متميزتين تماماً عند استخدامها في المصفوفات. فيمكنها تحديد طول السلسلة الحرفية ، كما هي الحال دائماً ، المحفوظة في عنصر المصفوفة وذلك على النحو التالي:

```
marray[2]         := 'This is a test'
?  len(marray[2] )                 // 14
```

إلا أن الوظيفة LEN( ) تتيح لك أيضاً تحديد العدد الإجمالي للعناصر في مصفوفة. ويجب استخدام اسم المصفوفة كمتغير لهذه الوظيفة:

```
local marray  :=  { "Sulaiman" , "Emad" , "Omar"  }
? len(marray)             // 3
```

وكما أشرنا سابقاً ، فعلى خلاف بقية لغات البرمجة الأخرى ، فإن كليبر يسمح لك بحفظ قيم من نوع آخر ضمن عناصر المصفوفة ذاتها. ويرجع كل من أمري TYPE( ) و VALTYPE( ) القيمة "A" عندما تمرر إليهما المتغير (أو أي تعبير آخر) يشير إلى مصفوفة.

```
local marray  :=  array(6)
marray[1]    := ' This is element one'
marray[2]    := 2.0
marray[3]    := DATE( )
marray[4]    := .T.
marray[5]    := { 1, 2, 3, }
? valtype(marray)               // A
? valtype(marray[1])            // C
? valtype(marray[2])            // N
? valtype(marray[3])            // D
? valtype(marray[4])            // L
? valtype(marray[5])            // A
? valtype(marray[6])            // U
```

وبما أنه لم يتم تأسيس العنصر السادس ، فإنه سيرجع النوع "U" من كلمة Udefined ، ولن يسبب هذا أية مشكلة في هذه الحالة. إلا أن استخدام هذا في الإصدارات السابقة من كليبر ، مثل Summer'87 يسبب مشكلة عويصة إذا أشرت إلى عنصر مصفوفة غير محدد. ولاداعي للقلق هنا إذ كما أشرنا سابقاً فإن كليبر 5.x سيؤسس كافة العناصر الموجودة في المصفوفة إلى قيمة صفر NIL ، لذلك ، فمع أنك لم تحدد قيمة واضحة للعنصر السادس ، فإن القيمة هنا سترجع إلى صفر.

تمرير المصفوفات وعناصر المصفوفة

تذكر أنك عندما أنشأت مصفوفة ما ، وعينتها إلى اسم متغير فإن ذلك المتغير يحتوي على إشارة إلى المصفوفة (بدلاً من المصفوفة ذاتها) لذلك فإنك عندما تمرر المتغير الذي يشير إلى مصفوفة ما لتنتقل إلى وظيفة أخرى فإنك إنما تمرر إشارة المصفوفة وهذا يعني أنك إذا غيرت أي عنصر من عناصر المصفوفة في الوظيفة الدنيا ، فإن هذه التغييرات ستنعكس على وظيفة الاستدعاء أيضاً.

ومع ذلك ، فإنك إذا مررت عنصراً ما داخل مصفوفة باتباع اسم المصفوفة برمز سفلي ، فإن قيمة هذه المصفوفة سترسل إلى الوظيفة. وإذا أجريت أية تغييرات على عنصر المصفوفة في الوظيفة الدنيا فإنها لن تظهر على وظيفة الاستدعاء.

ويبين المثال التالي ثلاثة أمثلة عن تمرير مصفوفة كاملة بالإشارة إلى الوظيفة دنيا ، على النحو التالي:

```
function main
local  myarray  := { 1, 2, 3, 4 }
test1(myarray)
?  myarray[3]              //  100
test2(myarray)
?  myarray[1]              //  1
test2(@myarray)
?  myarray[1]              // "A"
return  nil

static  function  test1(nums)
```

```
nums[3]   := 100
return nil

static function test2(nums)
nums := { "A", "B" , "C" ,  "D" }
return nil
```

إن الوظيفة MAIN( ) تمرر المتغير MYARRAY (والذي يشير إلى مصفوفة ذات أربعة عناصر) إلى وظيفة TEST1( ) الذي يغير العنصر الثالث من هذه المصفوفة إلى ١٠٠ والذي يظهر في الوظيفة الأعلى MYARRAY أيضاً.

ويشير المثال الثاني إلى الخفاء في الطريقة التي يعالج بها كليبر المصفوفات. إن الوظيفة TEST2( ) تنشيء مصفوفة حرفية ذات أربعة عناصر تحتوي على "A" و "B" و "C" و "D" ولن يجري أي تغيير على الإطلاق على المصفوفة المشار إليها باسم MYARRAY في وظيفة MAIN( ). ويحمي كليبر هذه المصفوفات بشكل تلقائي عند تمريرها إلى وظيفة دنيا. إلا أنه يمكن تجاوز هذه الحماية بأن تسبق المتغير بخيار "@" كما بينا في المثال الثالث. وسيمكن هذا التركيب اللغوي من تغيير الوظيفة الدنيا إلى وظيفة عليا.

وكما أشرنا سابقاً ، يتم تمرير عناصر المصفوفة الفردية بالقيمة. لذلك ، لن تستطيع الوظيفة التحكم بعنصر المصفوفة مباشرة ، بل بنسخة منها فقط. ويبين الجزء التالي من البرنامج هذه الفكرة بتمرير عنصر واحد من مصفوفة MyFunc( ) والذي يغيرها ولكن على أساس محلي فقط.

```
function main
local myarray := { 1, 2, 3, 4 }
myfunc(myarray[3] )
? myarray[3]              // still 3
return nil

function myfunc(num)
? num++          // 4
return nil
```

وقد تواجهك حالات تريد أن تغير فيها محتويات عنصر مصفوفة في وظيفة دنيا ، ثم ترغب أن يظهر هذا التغيير على المستوى الأعلى. ويحدث هذا غالباً عند استخدام مصفوفات ذات GETs. ونقترح في مثل هذه الحالات أن تمرر اسم المصفوفة والرمز السفلي على أنهما متغيران مستقلان كما يلي:

```
function main
local myarray := { 1, 2, 3, 4 }
myfunc(myarray, 3)
? myarray[3]              // changed to 4
return nil

function my
```

اعتبارات جدول الرموز

عند إنشاء أي من متغيرات الذاكرة من نوع PRIVATE أو PUBLIC يتم حجز ١٦ بايت في جدول الرموز وسيظهر هذا في حجم الملف التنفيذي. فعلى سبيل المثال إن إعلان ١٠٠ متغير ذاكرة سيضيف ١٦٠٠ بايت إلى حجم الملف التنفيذي.

وعلى النقيض من ذلك. يمكنك الآن إعلان مصفوفة تحتوي على ١٠٠ عنصر ويمكن تعيين مرجعها إلى اسم متغير واحد ، ولن يأخذ هذا العمل سوى ١٦ بايت فقط في جدول الرموز. وعند مقارنة العملين سترى الفارق الكبير في توفير الذاكرة الذي يمكن أن تحصل عليه باستخدام المصفوفات وعناصرها في جدول الرموز بدلاً من متغيرات الذاكرة ، إلا أنك إذا كنت قد عدلت عن استخدام كل من خياري إعلانات PUBLIC و PRIVATE كما اقترحنا سابقاً ، فلن يكون التوفير في جدول الرموز ذا قيمة تقريباً. بل وستبارك لنفسك على عدم استخدام تلك الإعلانات ، إلا أنك قد تحتاج إلى مزيد من الإقناع الآن لاستخدام مزايا المصفوفات. حسناً ، فلنبدأ إذن بذلك.

تستهل المصفوفات عملية البرمجة على وحدات ، فعلى سبيل المثال ، من السهل جداً إنشاء وحدة برنامج على شكل روتين للتفريق/التجميع بتحميل مصفوفة بدلاً من تأسيس سلسلة من متغيرات ذاكرة ، وسنبين هذا المفهوم من خلال أمثلة عليه لاحقاً.

حفظ المصفوفات/استرجاعها

إن المصفوفات من نوع البيانات ، لها سيئة متميزة واحدة إذا قورنت بأنواع البيانات التقليدية المعروفة (الحروف ، والتاريخ ، والأرقام ، و المنطق). فلا يحتوي كليبر على طريقة آلية لحفظها أو استرجاعها من ملفات الذاكرة.

إلا أن هذا لحسن الحظ ليس عبئاً كبيراً ، إذ يمكن كتابة وظائف محددة لحفظ مصفوفات واسترجاعها في ملفات نصوص ، وبما أننا نشجع استخدام المصفوفات ، فقد وضعنا كافة هذه البرامج المطلوبة على الأسطوانة المرفقة بهذا الكتاب ، فيرجى أخذ العلم بذلك.

تغيير حجم المصفوفات ديناميكياً

كان ما أعلنته من متغيرات في الإصدارات السابقة لكليبر Summer'87 هو ماسنلتزم به في البرنامج كله. فالمصفوفة التي أعلنت على أنها ٥٠ عنصراً مثلاً ستبقى كذلك ، حتى ولو حذفت أي عدد من العناصر منها. وقد سبب هذا الكثير من التركيز الذي لاداعي له، وكثيراً من الإزعاج ، بل غالباً ما اضطر المبرمجين إلى استخدام ملفات قواعد بيانات (يمكن تغيير حجمها) ، على حين أن المصفوفات هي أفضل بكثير من استخدام هذه الطريقة المزعجة.

ويقدم كليبر 5.x طريقتين جديدتين لتغيير طول المصفوفات:

- إضافة عنصر إلى آخر المصفوفة باستخدام الوظيفة ( )AADD ، إذ تضيف هذه الوظيفة عنصراً إلى آخر المصفوفة وتوسعها بمعدل عنصر آخر.

- ويبين المثال التالي استخدام الوظيفة ( )AADD ، ويجب ملاحظة التغيير الذي يطرأ على المصفوفة TEST بعد إضافتها إلى المصفوفة MYARRAY. وبما أن المصفوفة

MYARRAY[2] تشير بشكل رئيس إلى عنوان الذاكرة ذاته الذي تشير إليه TEST ، فإن أي تغيير على المصفوفة TEST سيظهر أيضاً على المصفوفة .MYARRAY[2]

```
function array07
local myarray := { } , test := { 1, 2, 3 }
scroll( )
? len(myarray)            // 0
aadd(myarray, "this will be the first element" )
? len(myarray )           // 1
aadd(myarray, test)
aadd(test, 400 )
? len(test)               // 4
? len(myarray[2] )        // 4
return nil
```

■ تغيير حجم المصفوفة باستخدام الوظيفة ( )ASIZE : تعتبر هذه الوظيفة الجديدة الفعّالة قادرة على تكبير المصفوفة أو تصغيرها لتصبح ذات طول محدد. فإذا حددت طولاً أكبر من الطول العادي ، فسيتم إضافة العدد اللازم من العناصر إلى نهاية المصفوفة وستعطى هذه العناصر جميعها قيمة الصفر NIL. أما إذا حددت طولاً أقصر من الطول الحالي فسيتم قطع عدد العناصر من النهاية وتصبح في وضعية ميتة مغناطيسياً. (يجب الانتباه إلى أن البيانات التي تحتويها العناصر الميتة لن يمكن استعادتها من جديد).

ويبين المثال التالي طريقة استخدام وظيفة حجم المصفوفة ( )ASIZE.

```
local myarray := { }, test := {1, 2, 3 }
? len(myarray)            // 0
asize(myarray, 3)         // adds three NIL elements
? len(myarray)            // 3
? myarray[3]              // NIL
asize(myarray, 0)
? len(myarray)            // 0-empty again
inkey(0)
//
? len(test)               // 3
asize(test, 10)           // adds seven NIL elements
? len(test, 2)            // 10
```

```
asize(test, 2)          // lops off eight elements, includeing the 3
? len(test)             // 2
? test[1]               // 1
? test[2]               // 2
? asize(test, 2)        // does nothing, since length is already 2
```

إن هذه الوظيفة المرنة الديناميكية تفتح آفاقاً جديده من إمكانيات البرمجة ، إذ أنها أولاً تمكن المبرمج من تنفيذ مايسمى بوظائف التطبيق الحقيقي وإذا قرنت هذه الوظيفة مع الإعلانات الساكنة STATIC فإنه يمكن كتابة وظائف فعالة وقوية يمكنها التعامل مع أغراض الصيانة الداخلية للبرامج والمتغيرات الشاملة على مدى البرنامج.

التكديس الجيد

تتطلب البرمجة الصحيحة أعمال صيانة وتنظيف جيدة بعد الانتهاء من عمليات البرمجة ، ويجب أن تتضمن أعمال الصيانة والتنظيف هذه كلا من: إعادة تجهيز المؤشر كما كان وحجمه ، وتجهيز اللون وضبطه كما كان ، ومحتويات الشاشة ، ومنطقة العمل وغيرها.

وسنبين فيما يلي كيف كان الإصدار السابق من كليبر ، وهو Summer'87 يقوم بهذه العمليات جميعها:

```
function myfunc
private oldcolor, oldscrn, oldrow, oldcol
oldcolor = setcolor( )                  && save color setting
oldscrn  = savescreen(0, 0, 24, 79)     && save screen contents
oldrow   = row( )                       && save current row
oldcol   = col( )               && save current column
*
* body of function
*
setcolor(oldcolor)                      && restore color
restscreen(0, 0, 24, 79, oldscrn)&& restore screen
@ oldrow , oldcol say ' '               && restore cursor position
return whatever
```

وكان هذا كافياً لأغراض المبرمجين عامة ، إلا أنه لدى توفر السواكن static المنتشرة على مدى الملف كله ، أصبحت هذه الطريقة لاتفي بالغرض (إن لم تقل إنها باطلة وملغاة) في

الإصدار الجديد من كليبر 5.x. فليس هناك مايمنع من تغيير محتويات أي من متغيرات الحفظ السابقة الذكر. وكذلك ، ليس هناك سبب وجيه يدعوك إلى حمل تلك المعلومات ضمن الوظيفة التي تقوم بها ، والمرة الوحيدة التي تحتاجها في الواقع هي عند الخروج من البرنامج.

ولنحاول الآن إعادة كتابة هذه الوظيفة للتنظيف والصيانة من جديد باستخدام سواكن على مدى الملف file-wide statics وطريقة الكبسلة encapsulation.

```
function myfunc
SaveEnv( )
//
// body of function
//
RestEnv( )
return Whatever
```

وسيكون لدينا في ملف برنامج PRG. مستقل كل من الأمور التالية:

```
static envstack_ := {}

// stack used by SaveEnv() and RestEnv()

//-------- manifest constants to delineate structure of nested arrays
#define ROW      1
#define COLUMN          2
#define CURSOR          3
#define COLOR           4
#define MAXROW          5
#define MAXCOL          6
#define BLINK    7
#define NOSNOW          8
#define SCREEN          9
/*
   Function: SaveEnv()
   Purpose:  Save current cursor row/column/size, color, and screen
*/
function SaveEnv()
aadd(envstack_, { row(), col(), setcursor(), setcolor(),    ;
              maxrow(), maxcol(), setblink(),          ;
              savescreen(0, 0, maxrow(), maxcol()) } )
return nil
```

```
/*
   Function: RestEnv()
   Purpose:  Restore cursor row/column/size, color, and screen
*/
function RestEnv()
local ele := len(envstack_)
/* preclude empty array, which would cause an array access error! */
if ele > 0
  //------- verify that video mode has not changed
  if (envstack_[ele, MAXROW] != maxrow() .or. ;
          envstack_[ele, MAXCOL] != maxcol())
    setmode(envstack_[ele, MAXROW] + 1, envstack_[ele, MAXCOL] + 1)
  endif
  //------- reset row/column position
  setpos(envstack_[ele, ROW], envstack_[ele, COLUMN])
  //------- reset cursor state
  setcursor(envstack_[ele, CURSOR])
  //----- reset color
  setcolor(envstack_[ele, COLOR])
  //------- reset previous blinkbit setting
  setblink(envstack_[ele, BLINK])
  //------ restore screen
  restscreen(0, 0, maxrow(), maxcol(), envstack_[ele, SCREEN])
  //----- truncate array by lopping off last element
  asize(envstack_, ele - 1)
endif
return NIL

//------- end of file SAVEENV.PRG
```

١ – إن المصفوفة الساكنة ENVSTACK تم إعلانها قبل الوظيفة الأولى. وهي تبدأ من عنصر الصفر ، إذ أننا سنضيفها في كل مرة نستدعي فيها الوظيفة ( )SaveEnv.

ولدى استخدام متغيرات على مدى الملف يجب إعلانها قبل الوظيفة الأولى ، أو قبل الإجراء الأول لنضمن أنها ستكون مرئية في ملف البرنامج كله. وعلى النقيض من هذا فإنك إذا أعلنتها ضمن وظيفة ما ، فإنها لن تكون مرئية إلا لتلك الوظيفة ، وهذا مالاتريد أن تفعله قطعاً.

٢- إن الملف الذي يحتوي على كل من الوظائف ( )SaveEnv و ( )RestEnv يجب تجميعها باستخدام الخيار N/ ، الذي يلغي التعريف الآلي للإجراء الذي يحمل اسم ملف البرنامج PRG. ذاته.

٣- إن ثوابت البيان manifest constants معرّفة بحيث تمثل كل عنصراً من عناصر البيئة التي يجب حفظها واسترجاعها كما كانت. وهذا أمر يتعلق بوضوح القراءة فقط فهو لذلك أمر اختياري.

٤- إن الوظيفة ( )SaveEnv تنشيء مصفوفة حرفية كما يلي:

```
{   row( ) ,  col( ) ,  setcursor( ) ,  setcolor( ) ,  ;
    savescreen(0,  0,  maxrow( ) ,   maxcol( ) ) }
```

وهي تحتوي على كل من مكانتي السطر والعمود ، وحجم المؤشر ، وتجهيز اللون ومحتويات الشاشة الحالية.

٥- ويتم إلحاق هذه المصفوفة بعد ذلك إلى نهاية المصفوفة ENVSTACK_ بحيث يزداد طولها بمعدل عنصر واحد ، بمعنى آخر: بعد الاستدعاء الأول لكل من وظيفتي ( )SaveEnv و ENVSTACK_ ستحتوي على عنصر واحد ، وستكون مصفوفة ذات خمسة عناصر ، على النحو التالي:

```
envstack_[1,  1] =  screen  row
envstack_[1,  2] =  screen  column
envstack_[1,  3] =  cursor   size
envstack_[1,  4] =  color     setting
envstack_[1,  5] =  screen  contents
```

٦- عند استدعاء الوظيفة ( )ResEnv يتم اختبار طول المصفوفة ENVSTACK_ للتأكد من أنها ليست فارغة. (وسيحدث هذا إذا إستدعيت الوظيفة ( )ResEnv مثل استدعاء ( )SaveEnv أولاً). ويجب أن تتذكر أنه ربما أن مصفوفة ENVSTACK_ قد أعلنت على أنها ساكنة ، فإنها ستحتفظ بقيمتها خلال مدة البرنامج كله. ولذلك فعند

إعادة إدخال ملف PRG. الذي يحتوي على كل من SaveEnv( ) و RestEnv( ) فإن المصفوفة ENVSTACK_ ستظل تحتوي على ماوضعناه فيها سابقاً.

فإذا لم تحتو المصفوفة ENVSTACK_ على أية عناصر فإننا سنمرر بقية الوظيفة لنتجنب أية أخطاء في محاولة الوصول إلى المصفوفة. وبما أن مصفوفات كليبر ذات أساس واحد فلن يكون هناك شيء مثل مصفوفة ENVSTACK_[0].

٧- إذا افترضنا أن المصفوفة ENVSTACK_ تحتوي على مصفوفة متداخلة واحدة أو أكثر ، فإن الوظيفة RestEnv( ) تحصل على معلومات من المصفوفة أو العنصر الأخير الموجود في نهاية المصفوفة تلك.

- يتم إرجاع مكان المؤشر كما كان باستخدام الوظيفة SETPOS( ).
- يتم إرجاع شكل المؤشر باستخدام الوظيفة SETCURSOR( ).
- يتم إرجاع تجهيز اللون كما كان باستخدام وظيفة SETCOLOR( ).
- يتم إرجاع محتويات الشاشة كما كانت باستخدام الوظيفة RESTSCREEN().

٨- وأخيراً ، تنهي الوظيفة RestEnv( ) عمل المصفوفة ENVSTACK_ بوظيفة الحجم ASIZE( ) بحيث تنهي هذه الوظيفة العنصر الأخير بشكل فعال والذي كان قد أدى دوره المطلوب منه.

وتجدر الإشارة إلى أنه يجب أن تتذكر أنه ليس أي من هذه المعلومات المحفوظة مرئية في وظيفة الاستدعاء ، وبدلاً من ذلك تصبح هذه المعلومات على شكل كبسولة إلى جانب الوظائف التي تحتاجها فقط (وهي SaveEnv( ) و RestEnv( )). ويعتبر هذا برمجة وحداتية (على شكل وحدات) حقيقية ، ولم يكن هذا سوى مجرد حلم فقط في إصدار كليبر السابق Summer'87 وذلك لعدم احتوائه على الإعلانات الساكنة.

حفظ المجموعات SET باستخدام المكدس Stack

بما أن الوظيفة الجديدة SET( ) تمكننا من التوصل إلى متغيرات المجموعات الشاملة فيصبح من السهولة جداً إنشاء وظائف تعتمد على التكديس ، تقوم بحفظ كل من هذه وإرجاعها:

```
static setstack_ := {}      // stack used by SaveSets() and RestSets()

/*
   Function: SaveSets()
   Purpose:  Save all SET variables onto the SET stack
*/
function SaveSets
local xx, settings_ := array(_SET_COUNT)
//------ loop through all SETs to build subarray
for xx := 1 to _SET_COUNT          // see SET.CH
  settings_[xx] := set(xx)
next
aadd(setstack_, settings_)
return nil

//------ end of function SaveSets()


/*
   Function: RestSets()
   Purpose:  Restore all SET variables from the SET stack
*/
function RestSets()
local xx
local settings_
local ele := len(setstack_)
//------ preclude empty array!
if ele > 0
  settings_ = setstack_[ele]
  //------ loop through subarray - assigning each SET as we go
  for xx = 1 to _SET_COUNT          // see SET.CH
    set(xx, settings_[xx])
  next
  //------ truncate master SET stack
  asize(setstack_, ele - 1)
endif
return nil
```

```
//----- end of function RestSets()
```

١- كما لاحظنا سابقاً ، أن المصفوفة الساكنة المسماة _SETSTACK يجب إعلانها قبل إعلان الوظيفة الأولى ، إذ أنها تبدأ فارغة ، ثم يضاف إليها في كل مرة تستدعى فيها الوظيفة SaveSets( ).

٢- تنشيء الوظيفة SaveSets( ) مصفوفة محلية فارغة اسمها _SETTINGS تعتبر مكاناً يحتوي على كافة متغيرات SET.

٣- نؤسس حلقة FOR..NEXT لكل متغير من متغيرات SET بالاعتماد على العداد المسمى _SET_COUNT والموجود في ملف الترويسة العادي STD.CH ونأخذ قيمة كل متغير من متغيرات SET باستخدام وظيفة SET( ) ونؤسس العنصر المناسب من المصفوفة _SETTINGS.

٤- عندما تكتمل الحلقة ، نضيف مصفوفة _SETTINGS إلى المصفوفة _SETSTACK.

٥- يجب أن تتأكد من أن المصفوفة _SETSTACK ليست فارغة ، كما هي الحال عندما تأكدت من أن كلاً من RestEnv( ) و RestSets ليستا فارغتين. فإذا افترضنا أنها ليست فارغة ، فإنها ستؤسس المصفوفة المحلية _SETTINGS بحيث تصبح على آخر عنصر من المصفوفة _SETSTACK.

٦- يجب أن تنفذ حلقة FOR. .NEXT لكل متغير من نوع SET. وتمرر الوظيفة SET( ) متغيرين : عداد الحلقات ، (والذي يشير إلى أي متغير SET سيتأثر من جراء هذه العملية) والقيمة (من المصفوفة _SETTINGS) التي يجب تحديدها.

٧- عند انتهاء الحلقة ، يوقف عمل المصفوفة _SETSTACK باستخدام وظيفة التحجيم ASIZE( ) ، ويصبح العنصر الأخير في وضعية خمود مغناطيسي وقبل أن نمضي قدماً في

هذه المناقشة نبين فيما يلي موجهين للمعالج الأولي وهما من الأوامر المفيدة. ينهي الأول عمل مصفوفة بقطع العنصر الأخير منها ، وهو أمر مفيد في: "آخر عنصر أدخل ، أول عنصر يخرج " الوظيفة التي تعتمد على التكديس.

```
#xtranslate Truncate( <a> ) => asize( <a>, len<a> - 1 )
```

أما الموجه الثاني فهو مفيد عندما ترغب في حذف عنصر ما من مصفوفة وإعادة تحجيمها بالشكل الذي تريد. وكما تعلم ، فإن الوظيفة ( )ADEL تقبل متغيرين اسم المصفوفة وعدد العناصر المراد حذفها. ويبين الجزء التالي من البرنامج مصفوفة قبل استخدام هذه الوظيفة ( )ADEL وبعد استخدامها:

```
function test
local a := { 1, 2, 3, 4 }
adel(a, 3)                              // A is now { 1, 2, 4, NIL}
```

وكما ترى ، عزيزي القارىء ، فإن العنصر الأخير في المصفوفة حالياً هو صفر NIL وفي كثير من الأحايين نريد حذف هذا العنصر (صفر).

إلا أن مالم تعلمه عن هذه الوظيفة ( )ADEL هو : أنها تعيد الإشارة إلى المصفوفة موضع السؤال. لذلك يمكننا كتابة الموجه على النحو التالي:

```
#xtranslate AKill( <a> , <e> ) => asize( adel(<a>, <e>, len(<a>) - 1)
```

لاحظ أن التركيب اللغوي مماثل تماماً للوظيفة ( )ADEL ، والفارق الوحيد بينهما هو أن الوظيفة ( )AKill هي أكثر فائدة لأنها تنهي المصفوفة باستخدام الوظيفة ()ASIZE.

المصفوفات المتداخلة Nested Arrays

ذكرنا سابقاً أن كل عنصر من عناصر مصفوفة ما قد يحتوي على مصفوفة أخرى داخله وقد يبدو هذا سبباً للاضطراب بعض الأحيان. أولاً ، دعنا نبين فائدته في المثال التالي:

```
local myarray[4]
myarray[1] := "Do not despair"
myarray[2] := "it's really not"
myarray[3] := "that difficult"
myarray[4] := { "to", "nest" , "array" , "within" , "arrays" }
```

وقد يبدو هذا التركيب مألوفاً ، فهو ماتستخدمه لإعلان مصفوفة حرفية وتأسيسها. بل إن المصفوفة Myarray[4] هي مصفوفة داخل ذاتها وتحتوي هذه المصفوفة على خمسة عناصر يمكن الإشارة إليها كما يلي:

```
? myarray[4, 1]// to
? myarray[4, 2]// nest
? myarray[4, 3]// arrays
? myarray[4, 4]// within
? myarray[4, 5]// arrays
```

إلا أنك إذا حاولت الآن الإشارة إلى المصفوفة MYARRAY[4] بذاتها فلن يحدث أي شيء هنا. كما هو موضح في المثال التالي:

```
? myarray[4]          // لن تتحطم ، ولكن في الوقت ذاته لن تعرض شيئا
```

ويجب تضمين الرمز السفلي الثاني لتبين العنصر الذي تشير إليه في المصفوفة MYARRAY[4]. ويجب الانتباه أيضاً إلى أننا بدأنا أولاً بإعلان مصفوفة MYARRAY[4]. وإذا أردت نسج مصفوفات بشكل متداخل ، كما هو مبين في المثال

الأخير فإن هذا الإعلان مقبول تماماً. إلا أنك إذا علمت منذ البداية أن كل عنصر في مصفوفتك يحتاج إلى عدة أبعاد فيمكن أن تعلن ذلك في مصفوفتك كما يلي:

```
local myarray[4, 2]
myarray[1]   := { 5,  4 }
myarray[2]   := { 3,  6 }
myarray[3]   := { 7,  2 }
myarray[4]   := { 1,  8 }
? myarray[1, 2] // 4
? myarray[4, 1] // 1
? myarray[2, 2] // 6
? myarray[3, 1] // 7
```

يعامل كل عنصر في هذه المصفوفة على أنه مصفوفة تحتوي على عنصرين ، وحاذر من التوقف المفاجيء لهذه المصفوفات (المتداخلة) ، ويبين البرنامج التالي عينة من هذه الأخطاء السهلة:

```
local myarray[2, 2]
myarray[1]   := "Frist element"
? myarray[1,  1]// boom!
```

يتغير العنصر الأول من هذه المصفوفة المتداخلة إلى سلسلة حرفية. وأما المحاولة التالية للإشارة إلى المصفوفة MYARRAY[1] كمصفوفة يسبب خطأ في أثناء وقت التشغيل إذ لم تعد المصفوفة مصفوفة.

تحذير
لاتستخدم الوظيفة AFILL( ) لملء المصفوفة بمصفوفة متداخلة.

```
ststic  function AfillDanger
local  myarray[100]
afill(myarray,  { 1 } )
?  myarray[1][1] // 1
```

```
myarray[1][1]     := 5
? myarray[5][1]
return nil
```

ويبين المثال التالي كيف تتداخل مصفوفتان بشكل يفقد الأمل من التعامل معهما:

```
function main
local a := { 1, 2, 3 }
local x := { 4, 5, a }
aadd(a, x)
inkey(0)
return nil
```

تحتوي المصفوفة A[4] على مصفوفة X[3]. و X التي تحتوي بدورها على مشير إلى المصفوفة A. لذلك ، فإن المصفوفة A[1] هي مماثلة للمصفوفة A[4][3][1]. حاول الآن تغيير هذه القيم في برنامج اكتشاف الأخطاء وتصحيحها Debugger لتتأكد من هذا بنفسك.

والآن ، لننظر إلى أمثلة واقعية أكثر إرباكاً وإزعاجا عن المصفوفات المتداخلة.

```
function test1
local myarray  := { { 5, 4 }, { 3, 6 }, { 7, 2}, { 1, 8} }

? myarray[1, 2]          // 4
? myarray[4, 1]          // 1
? myarray[2, 2]          // 6
? myarray[3, 1]          // 7
test2( )
return nil

function test2
local myarray := { 'element one', 2.0, date( ), .T. , { 3, 2, 1 } }
? myarray[1]             // element one
? myarray[2]             // 2.0
? myarray[3]             // today's date
? myarray[4]             // .T.
? myarray[5, 1]          // 3
? myarray[5, 2]          // 2
? myarray[5, 3]          // 1
aadd(myarray[5] , 0)
return nil
```

تعمل الوظيفة AADD( ) على مقربة من نهاية الوظيفة TEST2( ) لتغير حجم المصفوفة الموجودة في MYARRAY[5].

قائمة بوظائف المصفوفات في كليبر 5.x

تشبه وظائف المصفوفات في الإصدار الجديد لكليبر 5.x الوظائف في الإصدارات السابقة.

الوظيفة AADD( )

تضيف هذه الوظيفة عنصراً إلى نهاية المصفوفة وبهذا يتغير الحجم.

الوظيفة ACHOICE( )

تنفذ هذه الوظيفة قائمة أوامر منسدلة من الاختيارات المحفوظة في مصفوفة.

الوظيفة ACLONE( )

تعمل هذه الوظيفة على إعداد نسخة من مصفوفة متداخلة أو ذات أبعاد مختلفة وهو أمر مشابه لأمر النسخ ACOPY( ) إلا أنه أفضل منه بكثير لعدة أسباب: (أ) تعمل هذه الوظيفة على المصفوفات المتداخلة. و(ب) لاحاجة لإنشاء المصفوفة الهدف مسبقاً باستخدام هذه الوظيفة.

وهناك سبب وجيه آخر لاستخدام هذه الوظيفة وهو : لنفترض أنك تريد عمل نسخة من مصفوفة ما وتعمل على النسخة ، فإذا حاولت استخدام الطريقة التالية فإنك سترى أنك تغير المصفوفة الأصلية أيضاً.

```
function main
local x := { 1, 2, 3, 4, 5 }
local y := x
y[2]  :=  200
```

```
? x[2]          // is also 200
return nil
```

تضمن هذه الوظيفة أن Y لاتشير إلى عنوان الذاكرة على أنه X وهذا يعني أن العنوان Y ستكون مصفوفة مختلفة تماماً عن X.

```
function main
local x := { 1, 2, 3, 4, 5 }
local y  :=  aclone(x)
y[2]  := 200
? x[2]            // still 2
return nil
```

إذا أردنا إيضاح هذه النقطة أكثر ، فلننظر إلى المثال التالي ، حيث سيتم استدعاء وظيفة تحتوي على مصفوفة ساكنة ذات معلومات شاملة ، وكل عنصر من عناصر هذه المصفوفة الساكنة هو مصفوفة بذاته.

```
function main
local x := globals(1)
local y
aadd(x, 1)                // changes GLOBALS[1] in Globals( ) below!
y := globals(1) // this proves it
return nil

ststic function globals(item)
ststic globals := {       {},  ;
                          {},  ;
                          {},  ;
                          {} }
return globals[item]
```

والجزء الذي يزعج حقاً ويسبب الاضطراب ، هو أنه بعد أن تقوم بجمع أحد التجهيزات الشاملة وحفظها في عنوان الذاكرة X ، فإن التغيير إلى هذا العنوان X في الحقيقة ، يغير العنصر في المصفوفة الشاملة! وهذه شيفرة تحذيرية !.

الوظيفة ACOPY( )

تنسخ هذه الوظيفة العناصر من مصفوفة إلى أخرى إلا أنها لاتستطيع التعامل مع المصفوفات المتداخلة مما يجعلها عرجاء وقليلة النفع لمعظم العمليات في كليبر 5.x.

الوظيفة ADEL( )

تمسح هذه الوظيفة عنصر مصفوفة وتزيح كافة العناصر العليا درجة إلى الأسفل. وسيحتوي العنصر الأخير (أو الأعلى رقماً) في تلك المصفوفة ، قيمة الصفر NIL. وعادة ماتتبع هذه الوظيفة باستدعاء الوظيفة الأخرى ASIZE( ) لتقطع العنصر الجديد الذي قيمته صفر NIL من نهاية المصفوفة.

الوظيفة ADIR( )

تملأ هذه الوظيفة المصفوفات بمعلومات الدليل. (ملاحظة : لقد تم إلغاء هذه الوظيفة واستبدالها بالوظيفة DRECTORY( ) والتي سنناقشها بتفصيل لاحقاً).

الوظيفة AEVAL( )

تقيم هذه الوظيفة كتلة الشيفرة لكل عنصر من المصفوفة. وسنبين مزيداً من التفصيل عن كتلة الشيفرة لاحقاً.

الوظيفة AFIELDS( )

تملأ هذه الوظيفة المصفوفات بمعلومات تعريف الحقول (وقد تم استبدال هذه الوظيفة بالوظيفة DBSTRUCT( ) والتي سنصفها بعد قليل).

الوظيفة AFILL( )

تملء هذه الوظيفة بعض عناصر المصفوفة أو جميعها بقيمة مختارة.

الوظيفة AINS( )

تقحم هذه الوظيفة عنصراً قيمته صفر NIL في مصفوفة ، وتزيح كل العناصر الأعلى منه درجة واحدة. وتجدر الإشارة إلى أنها لاتغير طول المصفوفة إذ أن طولها لايتغير إلا باستخدام الوظيفة AADD( ) أو الوظيفة ASIZE( ).

وظيفة ASCAN( )

تمسح هذه الوظيفة المصفوفة الكترونياً بحثاً عن قيمة محددة ، أو حتى يتم تقييم كتلة الشيفرة على أنها حقيقية True. وتجدر الإشارة إلى أن هذه الوظيفة لن تعمل على المصفوفات المتداخلة ، مالم تستعمل كتلة الشيفرة.

الوظيفة ASIZE( )

تغير هذه الوظيفة حجم المصفوفة إلى حجم معين.

الوظيفة ASORT( )

تفرز هذه الوظيفة المصفوفات طبقاً لمحتوياتها (الترتيب المفترض هو التصاعدي). وتجدر الإشارة إلى أنك إذا أردت فرز محتويات مصفوفات متداخلة فلابد من تمرير كتلة شيفرة إلى هذه الوظيفة.

الوظيفة ATAIL( )

ترجع هذه الوظيفة البسيطة العنصر الأخير أو الأعلى في مصفوفة ما ، إلا أن لها مكانها لتساعدك على أن توقف البرمجة بالشكل التالي:

```
?  getlist[len(getlist)]                  // atail(getlist) قاعدة لغوية جديدة
```

وتبين الأمثلة التالية هذه الوظيفة:

```
a_  := { 1,  2,  3,  { "a",  "b",  "c"  }  }
? len(atail(a_) )              // 3
? atail(atail(a_) ) //  "c"
```

ثلاث وظائف أخرى للمصفوفات

تستخدم هذه الوظائف الثلاث للمصفوفات ، علماً بأنها ليست لمعالجتها إذ أنها تستخدم في المصفوفات المتداخلة. وهي ذات مزايا رائعة توزن بالذهب.

الوظيفة DBSTRUCT( )

تنشيء هذه الوظيفة مصفوفة ذات مصفوفات متداخلة تحتوي على معلومات تركيبية عن ملف قاعدة البيانات المستخدمه حالياً. وتحتوي المصفوفة على عنصر واحد لكل حقل في قاعدة البيانات. وسيكون كل من هذه العناصر مصفوفة من أربعة عناصر والتي تطابق الحقل حسب الترتيب التالي:

١- Name (الاسم)

٢- Type (النوع)

٣- Length (الطول)

٤- Decimals (الفواصل العشرية)

ويستدعي هذا الحاجة إلى مثال آخر ، ولنستخدم مثلاً ملفاً يسمى: INVOICE.DBF والذي يحتوي تركيبه على مايلي:

FieldName	Type	Length	Decimals
INV_NO	C	6	
CUST_NO	C	6	
DATE	D	8	
AMOUNT	N	10	2

```
#include   "dbstruct.ch"

function main(dbf_file)
local cfields,  x
use (dbf_file) new
cfields := dbstruct( )
? "Structure of " + dbf_file
? "Field Name  Type   Width    Decimals"
for x := 1 to len(cfields)
    ? padr(cfileds[x, DBS_NAME] , 12 ) , cfields[x, DBS_TYPE] , ;
      cfields[x , DBS_LEN] , cfileds[x , DBS_DEC]
next
return nil
```

وسيعطينا هذا البرنامج المخرجات التالية:

```
INV_NOC     6      0
CUST_NO     C      6      0
DATE        D      8      0
AMOUNT      N      10     2
```

(وتكون الأرقام متساوية من اليمين).

الوظيفة DBCREATE( )

تعتبر هذه الوظيفة متكاملة وتنسيقية للوظيفة السابقة ، فهي تنشيء بنية قاعدة بيانات من مصفوفة مصفوفات متداخلة تحتوي على معلومات عن الحقل. وتمكنك هذه الوظيفة من إعداد ملفات قواعد بيانات بمنتهي السهولة من خلال طريقة البرمجة المقترحة. ولعل أفضل

مثال على هذا هو إنشاء ملف قاعدة بيانات اسمه DBF. INVOICE من المثال السابق على النحو التالي:

```
dbcreate("invoice" , {     { "INV_NO" ,      "C" ,  6, 0 } ,  ;
                           { "CUST_NO" , "C",  6, 0 } ,  ;
                           { "DATE" ,        "D",  8, 0 } ,  ;
                           { "AMOUNT",      "N". 10, 2 }  }  )
```

وقد كانت مثل هذه العملية في إصدارات كليبر السابقة تحتاج كثيراً من البرمجة والزمن اللازم لتنفيذها وذلك لأنها تحتاج إلى ملف مؤقت يطلق عليه "توسيع البنية" (Structure extended).

ملاحظة لمستخدمي كليبر إصدار 5.2

تقبل هذه الوظيفة حالياً متغيراً ثالثاً وهو: <cDriver> ، فإذا تم تحديده سيصبح هو المستخدم لإنشاء قاعدة البيانات. وأما الطريقة المفترضة التلقائية فهي أن يستخدم DBFNTX.

ويبين المثال التالي كيف يمكن تعديل بنية قاعدة البيانات "على الطاير on-the-fly " باستخدام هذه الوظيفة والوظيفة السابقة. يمكنك فتح أية قاعدة بيانات وإضافة حقل تجريبي لها باسم DUMMY.

```
function  main(dbf_file)
local  a
use  (dbf_file)
a   :=  dbstruct( )
aadd(a ,   "DUMMY" ,  "C" ,  10,  0 }  )
dbcreate("new",  a)
use  new
append  from  (dbf_file)
use
// --------- look  for  existing backup files, and delete them if there
if  file(dbf_file  +  '.bak' )
   ferase(dbf_file  +  '.bak' )
endif
if  file(dbf_file  +  '.tbk' )
    ferase(dbf_file   + '.tbk' )
```

```
endif
frename(dbf_file + ".dbf" , dbf_file + ".bak" )
if file(dbf_file + ".dbt" )
   frename(dbf_file + ".dbt" , dbf_file + ".tbk" )
endif
frename('new.dbf' , dbf_file + ".dbf" )
if  file("new.dbt")
    frename('new.dbt' , dbf_file + ".dbt" )
endif
return nil
```

الوظيفة DIRECTORY( )

تعيد هذه الوظيفة مصفوفة تحتوي على مصفوفات متداخلة وتحتوي على معلومات عن دليل ما. وعلى غرار الوظيفة DBSTRUCT( ) سيكون هناك عنصر مصفوفة ذات عنصر واحد لكل ملف مطابق يبين مواصفات الملف. وسيصبح كل عنصر من عناصر هذه المصفوفة بدوره مصفوفة جديدة تحتوي على خمسة عناصر هي:

١- filename (اسم الملف)

٢- size (الحجم)

٣- date (التاريخ)

٤- time (التوقيت)

٥- attributes (المزايا ، مثل الأرشفة ، والإخفاء ، وغيرها)

ونبين فيما يلي مثالاً بسيطاً عن وظيفة DIRECTORY( ) عن كل شيء تحتويه مكتبة كليبر:

```
local files  := directory ( )
aeval(files ,  {  | a  |  qout(  padr(a[1],   14) ,  a[2] , a[3] ,  a[4] , a[5]  )  }  )
```

مخرجات هذه الوظيفة :

```
CLIPPER  LIB      510,097 02-15-93   5:20a
CLD      LIB       80,719 02-15-93   5:20a
DBFNTX   LIB       38,465 02-15-93   5:20a
DBFNDX   LIB       27,175 02-15-93   5:20a
DBFCDX   LIB      103,843 02-15-93   5:20a
DBFMDX   LIB       75,861 02-15-93   5:20a
DBPX     LIB      170,645 02-15-93   5:20a
EXTEND   LIB      125,881 02-15-93   5:20a
RTLUTILS LIB       53,925 02-15-93   5:20a
TERMINAL LIB       14,369 02-15-93   5:20a
ANSITERM LIB       12,321 02-15-93   5:20a
NOVTERM  LIB       13,857 02-15-93   5:20a
PCBIOS   LIB       14,369 02-15-93   5:20a
SAMPLES  LIB       53,891 02-15-93   5:20a
```

التعامل مع المصفوفات الشاملة مع المصفوفات الساكنة

يمكن تطبيق مفهوم المتغيرات الساكنة على مدى الملف على إدارة المتغيرات الشاملة. ويمكن بذلك تقليل الاعتماد على المتغيرات العامة Public والتي لاتساعد على البرمجة الوحداتية ، ومع أن المثال التالي سيركز على إدارة اللون ، فإن هذه المبادىء تنطبق على كافة أنواع المتغيرات الشاملة بما في ذلك متتالية هروب الطباعة escape sequences printer، واستخدام رقم التعريف وغير ذلك.

الطريقة القديمة

يستخدم معظم المبرمجين الذين يستخدمون إصدار كليبر Summer'87 الألوان باستخدام المتغير Public في أول برنامجهم على النحو التالي:

```
* MAIN.PRG
public c_normal, c_bold, c_enhanced, c_blink, c_msg, c_warning
if iscolor( )
        c_normal      = ' W/B'
        c_bold        = '+W/B'
        c_enhanced    = '+GR/B'
        c_blink       = '+W/B'
        c_msg         = '+W/RB'
        c_warning     = '+W/R'
else
        c_normal      = ' W/N'
        c_bold        = '+W/N'
        c_enhanced    = '+W/N'
```

```
        c_blink         = '+W/N'
        c_msg           = 'N/W'
        c_warning       = 'N/W'
ENDIF
....
```

وقد كان الخيار الآخر هو أن يضع المبرمج هذه الشيفرة في الوظيفة ( )ColorInit والتي ستستدعي بدورها لدى الدخول في البرنامج الرئيسي. وفي أية حالة من هاتين الحالتين فإن المتغيرات سيتم إعلانها على أنها PUBLIC بحيث يمكن رؤيتها من كل مكان في البرنامج.

فعندما يحين الوقت لتغيير اللون فيمكن أن يرسل المبرمج أحد المتغيرات العامة PUBLIC إلى الوظيفة ( )SETCOLOR كما هو واضح في المثال التالي في الوظيفة Err_Msg():

```
err_msg("File not available")
*
*
#include "box.ch"

function err_msg
parameter msg
private buffer, leftcol, oldcolor, oldrow, oldcol
oldrow   = row( )
oldcol   = col( )
leftcol  = int( 79 - len(msg) ) / 2 )
oldcolor  = setcolor(c_warning)           && PUBLIC color setting
buffer    = savescreen(11, leftcol, 13, 80 - leftcol)
@ 11, leftcol, 13, 80 - leftcol box B_SINGLE + ' '
@ 12, leftcol + 2 say msg
inkey(0)
** restore screen, cursor position, and color
restscreen(11, leftcol, 13, 80 - leftcol)
@ oldrow , oldcol say ' '
setcolor(oldcolor)
return .t.
```

إدارة الألوان في كليبر 5.x

كانت إعلانات PUBLIC أفضل الحلول المتوفرة لدينا كوسائل نتعامل معها في مثل هذه الحالات في ذلك الوقت: إلا أنه لدى توفر المتغيرات الساكنة STATIC لم يعد هناك حاجة

لاستخدام متغيرات PUBLIC بهذه الطريقة ، إلا أن هـذه المتغـيرات السـاكنة STSTIC أيضاً تكشف عن نقطتي عجز لاتشتمل عليهما المتغيرات العامة PUBLIC ، وهي:

- إن متغيرات Public تتطلب وضع قيمة في "جدول الرموز" ، وهذا يعـني أن براجمك سيتطلب المزيد مـن المسـاحة في الذاكـرة ، وسيتم تنفيـذه بشـكل أبطـأ. في حـين أن المتغيرات الساكنة STATIC هي أسرع من المتغيرات Public وذلك لأن المتغـيرات الساكنة (والمحلية Local) يتم حلها أثنـاء وقـت التجميـع ، وبهـذا فهـي لاتحتـاج إلى قيمة في "جدول الرموز".
- بما أن المتغيرات العامة Public مشاهدة ومرئية مـن كـل مكـان ، فهـي أيضـاً عرضـة للتغيير. ونحن نعتقد أن أجهزة الكمبيوتـر قـد وصلـت إلى حـد نسبي عـالٍ جـداً مـن كمال الأداء ، إلا أن المبرمج ذاته هو الذي تكمن المشكلة فيه ، فقد يرتكب بين آونة وأخرى غلطة حمقاء ، وخاصة عندما يكون مرهقاً مثلاً.

وليس هناك أسهل من مسح قيمة متغير عام PUBLIC بالخطأ ، وخاصة إذا لم تكـن تستخدم طريقة جيدة لتسمية المتغيرات التي تستخدمها في البرنامج. وعلى النقيض من هذا فإن كلاً من المتغيرات السـاكنة (والمحلية) لايمكن مشاهدتها إلا ضمـن الوظيفـة أو الإجراء التي أنشأتها فيه ، وهذا يعني أنك لن تستطيع مسحها بالخطأ أو تغييرهـا مـا لم تكن ضمن الوظيفة التي استخدمتها لإنشائها.

الكبسلة Encapsulatin

إن الحيلة التي يجب أن يعمد المبرمج إلى استخدامها هنا هي أن يضع متغـيرات اللـون بعيـداً عن أماكن الخطر. ولعل أفضل طريقة للقيـام بذلـك هـو أن تضعهـا داخـل "كبسـولة" مـع الوظيفة أو الوظائف التي استخدمتها لإعدادها واستخدامها أو التوصـل إليهـا. ونبـين فيمـا يلي مثالاً بسيطاً على طريقة الكبسلة المقترحة وهي كما يلي :

```
function C_Normal(newcolor)
ststic color  := "W/B"
/* change color setting if parameter was received */
if newcolor  != NIL
   color = newcolor
endif
return setcolor(color)
```

إن الوظيفة ( )C_Normal تعلن متغيراً ساكناً على أنه COLOR لأبيض على أزرق. وتقبل هذه الوظيفة متغيراً اختيارياً هو NEWCOLOR. فإذا تم تمرير هذا المتغير فإنه سيتم تحديد قيمة المتغير الساكن COLOR. ولعلك ترى الآن لماذا يجب أن نعلن COLOR على أنه متغير ساكن بدلاً من متغير محلي ، إذ نريد أن يحتفظ بقيمته للمرة التالية عندما نستدعي الوظيفة ( )C_Normal. ويجب أن تبدأ المتغيرات المحلية من جديد في كل مرة يتم فيه استدعاء هذه الوظيفة.

وأخيراً ، فإن الوظيفة ( )C_Normal تستدعي الوظيفة ( )SETCOLOR حتى تغير تجهيز اللون. وتعيد هذه الوظيفة قيمة إرجاع ( )SETCOLOR ، وهو للتجهيز الحالي للون ، بحيث يمكنك حفظ هذا في متغير واسترجاعه في مكان آخر.

وهناك طريقتان لتأسيس الوظيفة ( )C_Normal ، وإحداهما أن تستدعيها دون متغيرات مثل:

```
C_Normal( )
```

وستقوم هذه الوظيفة ببساطة بتغيير اللون إلى الأبيض على الأزرق (بافتراض أننا لم نغير اللون المفترض أصلاً). إلا أنه يمكن تغيير هذا الافتراض بإرسال متغير وتحديده كما يلي:

```
c_normal( "+w/r")
scroll( )
setcolor("w/b")
qout(" this  will be  white  on  blue")
c_normal( )
qout(" this will be  hi white  on  red")
```

وسيسبب هذا تغيير اللون المفترض إلى أبيض ساطع على أحمر. وبما أن قيمة محتويات ذاك المتغير ساكنة فإنها ستحتفظ بقيمتها مهما إستدعيت أمر C_Normal( ) كما هو مبين في المثال السابق.

وهذه الطريقة فعالة في إخفاء تجهيز اللون في الوظيفة التي تحتاج للتعرف عليه فقط. ولن يمكنك تغييره بشكل خاطىء ، ولكن يمكنك تغييره عند اللزوم إلا أنه يترتب عليك القيام بهذا عن تصميم وقصد بإرسال متغير خاص للقيام بذلك التغيير ، وسيصبح التغيير أكثر قابلية للتحكم به من الماضي.

أما إذا ظننت أن هذه الطريقة تتطلب المزيد من الجهد والعمل ، فإنك قد تكون نصف محقٌ في هذا ، إذ أنها تتطلب منك القيام بمزيد من التفكير أثناء كتابة برنامجك ، ولكن يجب أن تتذكر أنك تقضي وقتاً أطول بكثير أثناء القيام بعمليات صيانة أكبر بكثير من الوقت الذي تقضيه في التفكير أثناء كتابة هذا البرنامج بالشكل الصحيح المطلوب (أي أثناء اكتشاف الأخطاء وتصحيحها).

ونقترح أن تلتزم بكتابة برامجك بهذه الطريقة لأنها ستوفر عليك كثيراً من الوقت، كما يمكن أن تسهل عملية تغيير الألوان بالاحتفاظ بكل ماتريد في مكان واحد.

وظيفة واحدة وألوان عديدة

إن الوظيفة C_Normal( ) قد استخدمت هنا فقط لتمثيل عملية "الكبسلة". وقد يكون غير عملي تحديد وظيفة لكل لون. ولنأخذ وظيفة كليبر SET( ) كمثال على هذا.

يستخدم كليبر هذه الوظيفة للتعامل مع التجهيزات الشاملة (والتي كانت ٣٧ تجهيزاً). ولنراجع معاً التركيب اللغوي لهذه الوظيفة SET( ) فهو:

```
SET ( < setting > [ ,< newvalue> ] )
```

فإن المتغير الأول هو متغير رقمي يحدد التجهيز الأول الذي يجب البدء بالبحث عنه. ويمكن أن تجد قائمة كاملة بثوابت البيان لكل تجهيز من التجهيزات في ملف الترويسة المسمى SET.CH. فعلى سبيل المثال ، يمكن الإشارة إلى تجهيز لوحة المفاتيح CONSOLE باستخدام ثابت البيان التالي:

```
#define _SET_CONSOLE              17

oldcons = set(_SET_CONSOLE, .F.)   // set console off
*
*

set(_SET_CONSOLE, oldcons)         // restore previous value
```

تحذير

يجب الإشارة إلى التجهيزات دوماً باستخدام ثوابت البيان ، بدلاً من قيمتها الرقمية. وتحذر الحادات الكمبيوتر أن هذه القيم الرقمية هي عرضة للتغيير ، بينما يمكن الاحتفاظ بثوابت البيان manifest constants ، كما أن هذه أيضاً هي أسهل جداً لتذكرها والتعامل معها.

وتقبل الوظيفة SET() ، على غرار C_Normal( ) متغيراً اختيارياً هو <newvalue>، وإذا تم تمرير هذا المتغير فإن التجهيزات العامة ستتغير لتصبح هي أيضاً القيمة الجديدة الاختيارية ، فيرجى الانتباه.

وبناء على هذا الأمر ، سنحاول إعادة كتابة الوظيفة C_Normal( ) بحيث يمكنها أن تتعامل مع كافة الألوان بدلاً من لون واحد. والهدف من هذا التغيير هو استخدام مصفوفة ساكنة تحتوي على عدة تجهيزات لون ويمكن تحقيق ذلك بسهولة حسب الطريقة التالية:

```
function ColorSet(colornum, newcolor)
ststic colors := { "W/B" , "+W/B" , "+GR/B" , "*W/B" , "+W/RB" , "+W/R" }
/* change applicable color setting if second parameter was
passed */
if newcolor != NIL
   colors[colornum] := newcolor
endif
return setcolor(colors[colornum] )
```

ولنقم أيضاً بتأسيس ثوابت البيان بحيث لانحتاج لتذكر عنصر المصفوفة المطابق للون الذي نريد تغييره ، وذلك على النحو التالي:

```
#define     C_NORMAL       1
#define     C_BOLD         2
#define     C_ENHANCED     3
#define     C_BLINK        4
#define     C_MESSAGE      5
#define     C_WARNING      6
```

ويبين الجزء التالي من البرنامج كيف يمكننا الإشارة إلى هذه الألوان ، كما نستخدم أيضاً الوظيفتين السابقتين الذكر وهما ( )SaveEnv و ( )RestEnv للقيام بأعمال التنظيف والصيانة اللازمتين للبرامج المكتوبة.

```
#include "box.ch"
#include "colors.ch"
function missing( )
saveenv( )
ColorSet(C_BOLD)
@ 11, 30, 13, 49 box B_SINGLE + ' '
ColorSet(C_BLINK)
@ 12, 32 say "Record not found"
inkey(0)
restenv( )
return nil
```

النقاش الكبير: "الألوان المتعددة" مقابل "اللون الواحد"

لاشك أن الإجابة واضحة . فإن الله سبحانه وتعالى خلق لنا الألوان لنستمتع بها ، وللأسف الشديد ، فها نحن نشهد أفول عصر استخدام الشاشة الملونة العادية VGA لنشهد عصر استخدام شاشة جديدة وهي SVGA ، إلا أننا مع هذا كله نرى أن بعض مستخدمي الكمبيوتر لايزالون يستخدمون شاشة أحادية اللون ، أكل الدهر عليها وشرب ! ... فلابد إذن من ابتكار طريقة سهلة لمعالجة هذا الأمر.

يجب اتباع الخطوات التالية:

- اكتب وظيفة مصغرة لتأسيس تجهيز الألوان على على الألوان أو الأبيض والأسود.
- وسع المصفوفة الساكنة لتجهيز الألوان بحيث تتضمن مساويات اللون الأحادي.

ويجب إعلان متغير الألوان الساكن على مدى الملف بأكمله لتنفيذ الخطوة الأولى، ويعتبر هذا الأمر ضروريا إذ يجب أن يكون مشاهدا في وظيفتين.

إن وظيفة ()ColorInit تؤدي دورا صغيرا إلا أنه هام جدا وهو: تأسيس اللون على أن يكون إما "حقيقي" (أي: نعم سنستخدم الألوان) ، أو: "غير حقيقي" (أسود وابيض). فإذا تم استدعاء هذا دون متغيرات فإنه سيستخدم وظيفة ()ISCOLOR كأساس للألوان في المستقبل.

وبناء على ما تقدم معنا حتى الآن فإن وظيفة ()ISCOLOR ليست معصومة عن الخطأ. فعلى سبيل المثال: ستنظر وظيفة ()ISCOLOR إلى ما يحدث لبطاقة فيديو سي جي إي في كل يوم وترجع القيمة إلى "حقيقي" حتى ولو كانت الشاشة غير قادرة على عرض الألوان. لذا ، فقد تم إعداد هذه الوظيفة بحيث تقبل متغيراً اختيارياً إذا تم اجتيازه فسيتم تجاوز هذا المتغير المنطقي للوظيفة بحيث يقابل اللون الأحادي الألوان الكاملة. وهنا اختر "حقيقي.T." للألوان ، و "غير حقيقي.F." للأبيض والأسود.

```
function colorinit(override)
color  := if(override  == NIL, if(iscolor( ), 1, 2) , ;
          if(override,  1,  2) )
return nil
```

أما الخطوة الثانية فهي تعديل المصفوفة في وظيفة ()ColorSet بحيث تحتفظ باللون الأحادي لكل تجهيز من الألوان. ويجب تغيير هذه المصفوفة من مصفوفة أحادية البعد تحتوي على ستة عناصر ، بحيث تصبح مصفوفة تحتوي على ست مصفوفات فرعية تحتوي كل منها على عنصرين ، وسيحتوي العنصر الأول من كل مصفوفة فرعية على تجهيز ألوان ، بينما يحتوي العنصر الثاني على تجهيز للون الأحادي. وسيقوم المتغير الساكن Color بدور المؤشر في المصفوفة الفرعية على النحو التالي:

```
function ColorSet(colornum, newcolor)
static colors := {       { "W/B",    "W/N"   } , ;
                         { "+W/B",  "+W/N"   } , ;
                         { "+GR/B", "+W/N"   } , ;
                         { "*W/B" ,  "*W/N"  } , ;
                         { "+W/RB", "N/W"    } , ;
                         { "+W/R" , "N/W"    } , ;
// change color setting if second parameter  was passed
if  newcolor  != NIL
   colors[colornum,  color]  := newcolor
endif
return  setcolor(colors[colornum,  color])
```

ويبين البرنامج التالي كيفية استخدام هذه الوظائف في برامجك.

```
#include"box.ch"
function  main
/*  verify  that this  is  REALLY a color system  */
if  iscolor( )
   qout("Press  C  for  color  monitor, any  other key  for mono")
   ColorInit(  chr(inkey(0))  $  "cC" )
else
  ColorInit(C_NORMAL)
scroll( )
ColorSet(C_BOLD)
@  11,  24,  13,  55  box  B_SINGLE  +  '  '
ColorSet(C_ENHANCED)
@ 12, 26  say "Welcome to the Brownout Zone"
inkey(0)
return nil
```

حفظ التغييرات

إن آخر قطعة من هذا اللغز هي كيفية حفظ تجهيز الألوان إذا تم تعديلها. ولايحتوي كليبر على برنامج جاهز يستخدم لحفظ المصفوفات وإعادتها كما كانت قبل التغيير إلا أنك ستجد على أسطوانة الكتاب المرفقة برنامجا صغيرا يمكنك من تحقيق هذه العملية. وتسمى هذه العمليات GSaveArray() و GLoadArray() وسنعتمد على هاتين الوظيفتين لحفظ تجهيزات الألوان وإرجاعها كما كانت عليه قبل التغيير.

كما يجب أن نجري تعديلين طفيفين على الوظائف الموجودة لدينا. أولا: يجب تعديل وظيفة ()ColorInit بحيث تقبل اسم ملف ، فإذا استقبلت اسما يجب أن تكون من الذكاء بحيث تحاول تحميل تجهيز الألوان من ذلك الملف المسمى لها ، وذلك على النحو التالي:

```
// convert logical value to 1 (.T.) or 2 (.F.)
#translate Logic2Num( <a> ) => ( if( <a>,  1, 2 )  )

function ColorInit(override)
local temparray
do case
  case override == NIL
    color := logic2num(iscolor())
  case valtype(override) == 'L'
    color := logic2num(override)
  case file(override)
    if len( temparray := gloadarray(override) ) == 0
      qout("Could not load colors from " + override)
      inkey(0)
    else
      colors := temparray
      color := logic2num(iscolor())
    endif
endcase
return NIL
```

وبما أن مصفوفات الألوان يجب أن تكون مرئية الآن ضمن وظيفة ()ColorInit وكذلك في ()ColorSet فيجب سحبها من الثانية وجعلها على مستوى الملف جميعه (ولعلك كنت تتوقع مثل هذا الإجراء على أي حال).

وإذا أردنا إكمال هذا السيناريو فقد أعددنا وظيفة ()ColorMod وهي تعرض عينات لكل تجهيز من تجهيزات الألوان المختلفة وتسمح لك بتغييرها بالشكل الذي تريده. ثم ستعطى خيار حفظ تجهيزات الألوان إلى ملف:

```
/*
  Function: ColorSet()
  Compile:  clipper colorset /n /w
  Purpose:  Demonstrate Clipper 5.0 color management with a file-wide
          static array and various accessor functions

  To create the demo, type "RMAKE COLORSET"
```

```
*/

#include "box.ch"
#include "inkey.ch"
#include "setcurs.ch"

#define TESTING       // to compile test stub

#define C_NORMAL          1
#define C_BOLD            2
#define C_ENHANCED        3
#define C_BLINK           4
#define C_MESSAGE         5
#define C_WARNING         6

#define COLOR_CNT         6

//----- default name for color configuration file - change if you want
#define CFG_FILE    "colors.cfg"

//----- convert logical to numeric: 1 if .T., 2 if .F.
#xtranslate Logic2Num( <a> ) => ( if( <a>, 1, 2) )

static color := .t.   // global flag for color (1) or mono (2)

/*
  the following array contains color and monochrome settings for each
  type of color.  The third array describes the color it applies to,
  which makes it completely self-documenting.  This third element is
  also used during the GColorMod() routine to identify each color.
*/
static colors := { { "W/B",   "W/N" , "Normal"  }, ;
                   { "+W/B",  "+W/N", "Bold"    }, ;
                   { "+GR/B", "+W/N", "Enhanced" }, ;
                   { "*W/B",  "*W/N", "Blinking" }, ;
                   { "+W/RB", "N/W" , "Messages" }, ;
                   { "+W/R",  "N/W" , "Warnings" } }


#ifdef TESTING   // begin test stub

function colortest
local oldcolor
local oldcursor
scroll()
do case

  case file(CFG_FILE)
    GColorInit(CFG_FILE)
```

```
  //----- verify that this is REALLY a color system
  case iscolor()
    qout("Press C for color monitor, any other key for monochrome")
    GColorInit( chr(inkey(0)) $ "cC" )
  otherwise
    GColorInit()
endcase
oldcolor := GColorSet(C_NORMAL)
oldcursor := setcursor(0)
scroll()
GColorSet(C_BOLD)
@ 11, 24, 13, 55 box B_SINGLE+' '
GColorSet(C_ENHANCED)
@ 12, 26 say "Welcome to the Brownout Zone"
inkey(3)
GColorMod()
GColorSet(C_BLINK)
@ 12, 26 say "Hope you enjoyed your visit!"
inkey(3)
setcolor(oldcolor)
setcursor(oldcursor)
scroll()
return nil

#endif  // end test stub


/*
   GColorInit(): initializes color management system
              to either color or monochrome, or
              load previously saved color settings
*/
function GColorInit(override)
local temparray
do case
  case override == NIL
    color := logic2num(iscolor())
  case valtype(override) == 'L'
    color := logic2num(override)
  case file(override)
    if len( temparray := gloadarray(override) ) == 0
      qout("Could not load colors from " + override)
      inkey(0)
    else
      colors := temparray
      color := logic2num(iscolor())
    endif
endcase
```

```
return NIL

* end function GColorInit()
*---------------------------------------------------------------------*

/*
   GColorSet(): changes color in accordance with
            internal settings stored in array
*/
function GColorSet(colornum, newcolor)
//----- modify color setting if second parameter was passed
if newcolor != NIL
   colors[colornum, color] := newcolor
endif
return setcolor(colors[colornum, color])

* end function GColorSet()
*---------------------------------------------------------------------*

/*
   GColorMod() - View/Modify all global color settings
*/
function GColorMod()
local key := 0, newcolor, ntop, xx, getlist := {}, colorfile
local oldscore := set(_SET_SCOREBOARD, .f.)  // shut off scoreboard
SaveEnv()
GColorSet(C_NORMAL)
ntop := ( maxrow() - COLOR_CNT ) / 2
@ ntop, 22, ntop + COLOR_CNT + 1, 57 box B_SINGLE + ' '
setpos(ntop, 0)
//----- pad each color setting to 8 characters for data entry
aeval(colors, { |a,b| colors[b, color] := padr(colors[b, color], 8) } )
for xx := 1 to COLOR_CNT
   @ row() + 1, 24 say colors[xx, 3] + " Color"
   GColorSet(xx)
   @ row(), 42 say "SAMPLE" get colors[xx, color] valid redraw(ntop)
   GColorSet(C_NORMAL)
next
setcursor(3)
read
setcursor(0)
//----- trim each color setting
aeval(colors,{ |a,b| colors[b, color] := trim(colors[b, color]) } )
setpos(ntop + COLOR_CNT + 1, 24)
dispout("Press F10 to save these settings")
if inkey(0) == K_F10
   colorfile := padr(CFG_FILE, 12)
   GColorSet(C_MESSAGE)
   @ 11, 18, 13, 61 box B_DOUBLE + ' '
```

```
  @ 12, 20 say "Enter file name to save to:"
  @ 12, 48 get colorfile picture '@!'
  setcursor(SC_NORMAL)
  read
  setcursor(SC_NONE)
  if lastkey() != K_ESC .and. ! empty(colorfile)
    gsavearray(colors, ltrim(trim(colorfile)))
  endif
endif
RestEnv()
set(_SET_SCOREBOARD, oldscore)
return NIL

* end function GColorMod()
*--------------------------------------------------------------------*


/*
  Redraw() - redraw color samples after each GET
*/
static function redraw(ntop)
local oldcolor := GColorSet(row() - ntop)
@ row(), 42 say "SAMPLE"
setcolor(oldcolor)
return .t.

* end static function Redraw()
*--------------------------------------------------------------------*

//----- end of file COLORSET.PRG
```

كتل الشيفرة Code Blocks

لقد أضاف كليبر 5.2 نوعين جديدين من البيانات إلى لغته ، وهما: NIL وكتل الشيفرة. وقد سبق لنا أن بينا NIL والتي تعتبر رائعة للتأكد من المتغيرات. وأما كتل الشيفرة فهي مفيدة للغاية إلا أنها قد وصفت – خطأ – بأنها صعبة الفهم جدا.

ويهدف النقاش التالي إلى تحقيق غرضين:(أ) تبسيط فكرة كتل الشيفرة بتسليط مزيد من الأضواء عليها بحيث يمكن فهمها والتعامل معها بيسر وسهولة ، (ب) إعطاء فكرة عن كيفية استخدامها ، ومتى يتم استخدامها ، بحيث تصبح بعد ذلك قادراً على فهمها والتعامل معها واستخدامها في برامجك بمنتهى السهولة والوضوح.

إن كتل الشيفرة جزء أساسي من كليبر لايمكن التغاضي عنه أو تجاهله. وحتى إذا لم تستخدم كتل الشيفرة على الإطلاق في برامجك فإن المعالج الأولي سيقوم بتحويل كثيرا من أوامرك إلى كتلة الشيفرة ، فلذا ننصحك بدراستها ومحاولة فهمها بشكل جيد.

تحتوي كتل الشيفرة على شيفرة مجمعة بلغة كليبر ، ويمكن تجميعها إما أثناء وقت التجميع مع بقية برامج كليبر ، أو أثناء وقت التشغيل باستخدام عامل"&".

ويعتبر المثال التالي أبسط أشكال كتل الشيفرة:

```
{ | [<argument list>] | <expression list> }
```

وتشبه كتل الشيفرة إلى حد كبير المصفوفات الحرفية لكليبر ، إذ تبدأ كلتاهما بقوس متعرج ({) وتنتهي بنظيره. إلا أن كتل الشيفرة تميز بوضع عمودين (| |)I بعد القوس المتعرج الأول مباشرة. وتفصل هذه الأعمدة قائمة متغيرات خيارية <argument list> تمرر بعد تقييمها إلى كتلة الشيفرة ، ويجب أن تنفصل عناصر هذه القائمة بفواصل فيما بينها.

وننصح باستخدام المسافة بين العمود والقوس المتعرج ، علما بأن وجودها هو محض أمر اختياري وجمالي ، إلا أنه يستحسن وجود المسافة لتسهيل القراءة.

ويعتبر تعبير <expression list> قائمة فصل بفاصلة لأي تعبير من تعابير كليبر، ويمكن تنفيذ هذه السلسلة كما سترى ذلك بنفسك لاحقا. وستجد أيضا ، ولحسن الحظ ، أن معظم أوامر كليبر لها وظائف مكافئة ، لذا يمكن استدعاء الوظيفة بدلا من استخدام الأوامر.

كتل الشيفرة هي وظيفية

تعتبر كتل الشيفرة بمثابة وظائف ، فبدلا من استدعاء كتلة شيفرة ، كما تستدعي الوظيفة، فيمكنك تقييمها فقط. ولننظر إلى وظيفة بسيطة ، وكتلة شيفرة لنلاحظ العلاقة المباشرة بينهما كما في المثال التال:

```
b := { |  | 50 }
? eval(b)
? myfunc( )

function myfunc
return 50
```

ومن الواضح أن نرى أن وظيفة MyFunc() سترجع الساكن ٥٠ ، وهو بالذات ما سترجعه لكتلة الشيفرة B ، وأما وظيفة كليبر EVAL() فتستخدم لتقييم كتلة الشيفرة ، وهذا يعني بشكل أساسي عملية استدعاء عادية.

ويمكن تقييم كتل الشيفرة باستخدام عدة وظائف تقييم مثل: EVAL() و AEVAL() و DBEVAL() ، كما يمك تقييمها داخليا أيضا باستخدام وظائف محددة مثل:ASCAN() و ASORT() ، وسترجع كتل الشيفرة لدى تقييمها أقرب قيمة صحيحة للتعبير الذي تحتويه. فعلى سبيل المثال: إذا أنشأت كتلة الشيفرة التالية:

```
local myblock := { |  | mvar }
```

فإن قيمة MVAR لن تكون مبرمجة داخل كتلة الشيفرة لدى إنشائها ، وبالطبع فإن قيمتها ستتغير أثناء تنفيذ برنامجك ، وكذلك في كل مرة تقيم فيها كتلة الشيفرة فإنها

سترجع القيمة الحالية لمتغير MVAR فإذا لم يكن هذا المتغير قد تم تحديده لدى تقييم كتلة الشيفرة فسيصدر لك كليبر رسالة خطأ: "متغير غير محدد undefined variable.

```
local myblock := { | | mvar } , mvar := 500, x
x := eval(myblock)
? x                          // output: 500
```

تذكر أن كتلة الشيفرة يمكن أن تحتوي على أية تعابير من التعابير المقبولة في كليبر. وهذا يعني أنه بإمكانك الاستفادة منها إلى أقصى الحدود ، وذلك للوفاء باحتياجاتك، على سبيل المثال:

```
local myblock :={ | | qout(var1), qqout(var2), 500 }
local var1 := "Clipper ", var2 := "5.x"
local x := eval(myblock)          // "Clipper 5.x"
? x                                  // 500
```

انظر ثانية إلى العبارة الأخيرة في كتلة الشيفرة السابقة ، كيف تحصل X على القيمة ٥٠٠؟ إنك لدى تقييم كتلة شيفرة باستخدام EVAL() فسترجع قيمة آخر تعبير موجود فيها (أو أقرب قيمة صحيحة له). وبما أن آخر تعبير كان ٥٠٠ فإنها ستفترض أنها هي القيمة المطلوبة.

ويجب الانتباه إلى استخدام وظيفة QOUT() بدلا من استخدام الأمر ؟ وذلك لأن المعالج الأولي غير قادر على ترجمة موجهات #command إذا تم استخدامها داخل كتلة الشيفرة. ولهذا السبب بالذات فإن من الأهمية بمكان أن تعرف الأوامر التي يتم تحويل وظائف كليبر إليها ، وإن الوظائف التي تراها مفيدة جدا في كتل الشيفرة هي الإخراجات البدائية وكافة الوظائف المتعلقة بقواعد البيانات.

استخدام كتل الشيفرة دون متغيرات

نقدم فيما يلي أمثلة على كتل شيفرة بسيطة لاتستخدم متغيرات. ويبين المثال الأول إخراج قيمة على الشاشة:

```
local myblock := { | |  qout(mvar) } , mvar := "testing"
eval(myblock)                          // "testing"
```

يرجع هذا المثال قيمة الثابت (٥٠٠٠) والتي سيتم تعيينها إلى المتغير X عند التقييم.

```
local myblock := { | | 5000 }
x := eval(myblock)
? x                        // 5000
```

يتوقف مثال التقييم التالي ، وذلك لأن المتغير X لم يتم تحديده.

```
local myblock := { | | x++ }
local y
for y := 1 to 100
    eval(myblock) // boom!
next
? x
```

يبين هذا المثال الطريقة الصحيحة لكتابة المثال السابق حيث تم تحديد قيمة المتغير X بحيث تم تنفيذ البرنامج بنجاح.

```
local myblock := { | | x++ }
local x := 1
local y
for y := 1 to 100
    eval(myblock)
next
? x
```

يبين هذا المثال طريقة استدعاء إحدى وظائفك الخاصة من داخل كتلة الشيفرة.

```
local myblock := { | | BlueFunc( ) }
eval(myblock)              // calls BlueFunc( ) which displays a message
return nil

static function bluefunc
? "here we are in a BlueFunc( ) - will we ever escape?"
inkey(5)
return nil
```

استخدام كتل الشيفرة بمتغيرات

لاشك أن هناك قوة كبيرة جدا يمكن استغلالها من خلال طريقة كتل الشيفرة ، كما هي الحال تماما باستخدام "الوظائف" المختلفة لبرمجة كليبر. وإن كتابة متغيرات لكتلة الشيفرة هي عملية مماثلة تماما تقريبا لكتابة متغيرات الوظيفة. إلا أننا سنحاول هنا كتابة كتل شيفرة مبسطة ، ثم نعيد كتابتها على شكل وظائف لتستطيع المقارنة بينهما:

```
local myblock  := {  | a, b, c |   max(a,  max(b,  c) )  }

function mmax(a, b, c)
return max(a,   max(b,  c) )
```

وكما يبدو لك من الوهلة الأولى ، ترجع وظيفة MMax() أعلى المتغيرات الثلاثة التي تم تمريرها إليها. وإن تقييم كتلة الشيفرة سيرجع الشيء ذاته تماما ، إلا أنه لابد من تجاوز حجر عثرة آخر وهو: كيف يمكن تمرير متغيرات إلى كتلة الشيفرة؟ إن الأمر بمنتهى البساطة ، إذ أن وظيفة EVAL() تقبل المتغيرات الاختيارية بعد اسم كتلة الشيفرة. ويمثل كل متغير اختياري المتغير المراد تمريره إلى كتلة الشيفرة. فعلى سبيل المثال: إذا كتبت:

```
eval(myblock,  20)
```

فإنك تمرر المتغير الرقمي ٢٠ إلى كتلة الشيفرة المسماة .MYBLOCK لننظر مرة ثانية على وظيفة MMax() وكتلة الشيفرة بحيث تتعرف بشكل جلي على كيفية تمرير المتغيرات باستخدام الوظيفة EVAL() :

```
local myblock := {  |  a,  b,  c  |  max(a,  max(b,  c))  }
?  mmax(20,  100,  30)                    // 100
?  eval(myblock, 20,  100,  30)           // 100
```

هل تذكر وظيفة BlueFunc() التي تحدثنا عنها آنفا؟ فلنحاول تعديل هذه الوظيفة وكتلة الشيفرة بحيث تقبل المتغير الذي سيملي كم ستكون فترة انتظار الضغط على لوحة المفاتيح.

```
local myblock  := { | x | BlueFunc(x)  }
eval(myblock, 20)                    // calls BlueFunc( ) and will wait 20 seconds
return nil

static function bluefunc(delay)
? "we're in a BlueFunc( ) for " + ltrim(str(delay)) + " seconds"
inkey(delay)
return nil
```

وإليك الآن كتلة شيفرة تقبل ثلاثة متغيرات كحد أعلى وتعرضها جميعها على الشاشة:

```
local myblock := { |  a, b, c | qout(a, b, c) }
eval(myblock, 1, 2, 3)              // 1  2  3
x := eval(myblock, 1, 2) // 1  2  NIL
? x                                 // NIL
```

لاشك أنك تعلم الآن لماذا تعرض العبارة التالية لوظيفة EVAL() كلا من القيم ١ ، ٢ ، وصفر أليس كذلك؟ إن هذا يرجع لأن أية متغيرات معلنة لم يتم استقبالها ستؤسس على أن قيمتها (0). وبما أن كتلة الشيفرة MyBlock تتوقع ثلاثة متغيرات هي: a, b, c) وقد أرسلنا متغيرين فقط ، فإن المتغير c سيؤسس على أنه (0). ولعلك تدرك أن كتلة الشيفرة ترجع قيمة التعبير QOUT(a,b,c) وهذه الوظيفة ترجع قيمة (0) دائما ، فيرجى الانتباه إلى هذه النقطة الدقيقة.

ملاحظة هامة

تعطى أي معادلة تحددها في كتلة شيفرة المعاملة المحلية بشكل آلي ، ولن تكون هذه المعادلات مرئية لأية مجموعات كتل شيفرة متداخلة. ولاشك أن هذا الأمر يحتاج إلى توضيح بمثال آخر كما يلي:

```
local firstblock  := { |  | qout(x)  }
local myblock  := { | x |  x++, eval(firstblock) }
eval(myblock,  3)
```

وسيتوقف هذا البرنامج عندما تحاول تقييم وظيفة FirstBlock() ، وقد يبدو أن المعادلة X في وظيفة MyBlock() يجب أن تكون مرئية في وظيفة FirstBlock() ويجب ألا تنخدع بهذه الفكرة إذ أن X هي محلية لوظيفة MyBlock() لذلك لن تكون مرئية بالنسبة لوظيفة FirstBlock() إذ يجب تمريرها بشكل واضح وصريح ومباشر من كتلة شيفرة إلى أخرى كي يمكن رؤيتها.

تجميع كتل الشيفرة أثناء وقت التشغيل

لقد تم حل عينات كتل الشيفرة التي قدمناها حتى الآن أثناء وقت التشغيل ، إلا أن هناك حالات لاتعرف ماذا يجب عليك أن تضع في كتلة الشيفرة إلى أن يحين وقت التشغيل. وقد يكون من خير الأمثلة على هذا "حالة تصفية" أو حركة كتل متعلقة باستعراض عنصر ما.

وإذا أردت السماح بحدوث مثل هذا ، يمكن تجميع مصفوفة حرفية إلى كتلة شيفرة أثناء وقت التشغيل. ويجب أن يكون لتلك المصفوفة الحرفية التركيب اللغوي ذاته الذي يستخدم لكتلة البرمجة ، كما يجب أن تكون محاطة بأقواس ، ومسبوقة بعلامة عامل & المستخدمة للماكرو ، علما بأنها تختلف في الأداء عنها.

إن إحدى الحالات التي قد يفيد فيها تجميع كتل الشيفرة أثناء وقت التشغيل هي عند إنشاء شروط تساؤل. وقد كان من العادي جدا أثناء استخدام كليبر Summer '87 إنشاء مصفوفة حرفية تحتوي على شرط تساؤل ثم يطبق عليها عامل الماكرو ، على النحو التالي:

```
condition := "cust->state == 'OR' "
do while ! eof( )
    if &condition
       *
    endif
    skip
enddo
```

وأما في كليبر 5.x ، فمن الأفضل إنشاء كتلة شيفرة ، ثـم تقييمهـا في كـل مـرة ، علـى النحو التالي:

```
condition := "cust->state == 'OR' "
bcondition := &("{ | | " + condition + "}")
do while ! eof( )
    if eval(bcondition)
       *
       endif
       skip
enddo
```

ويجب أن تتأكد من إنشاء كتلة الشيفرة قبل حلقة DO WHILE فانتبه لهذا تماما وإلا فإنك ستقع في ورطة لها بداية ، وليس لها نهاية... ربنا يستر!!

الماكرو في كتل الشيفرة

تجدر الإشارة إلى أننا لسنا من أنصار استخدام الماكرو ، ولكننا تلقينا العديـد مـن أسـئلة المبرمجين تدور حول معرفة كيفية معاملة الماكرو داخل كتل الشيفرة ، مما يدعونـا لشـرح هذه النقطة بشيء من الإسهاب والتفصيل للذكرى فقط!

تعامل الماكرو ضمن كتـل الشـيفرة بإحدى طريقتـين: "المبكـرة" أو "المتـأخرة". فالتعبير المبكر يعني أن الماكرو سيتوسع لدى إنشاء كتلة الشيفرة. ويبين البرنـامج التـالي أن قيمة FNAME قد تمت برمجتها في كتلـة الشـيفرة في تلـك اللحظـة بـالذات ، ولـن تعكس أية تغييرات يتم إجراؤها لاحقا.

```
private fname := "var1"
private var1 := 1
private var2 := 2
b := { | | &fname }
? eval(b)                    // output: 1
fname := "var2"
? eval(b)                    // output: still 1
```

إن التوسع المبكر هو إلى حد كبير مثل تحديد المصفوفات الحرفية تماماً. ويمكن إعادة كتابة المثال السابق على النحو التالي:

```
b := &("{ | | " + fname + "}")
```

إلا أنك إذا أردت استخدام ماكرو في كتلة شيفرة يمكن أن تتغير خلال برنامجك ، فقد ترغب بالاعتماد على التوسع " المتأخر" ، إذ تتم توسعة الماكرو في هذه الحالة عند كل مرة يتم فيها تقييم كتلة الشيفرة المحددة. ويعتبر الفارق في التركيب اللغوي بينهما خفيفاً جداً ، إذ يجب أن يوضع التعبير المراد توسيعه بالماكرو ضمن قوسين فقط. ونبين فيما يلي المثال السابق ذاته بعد إعادة كتابته بحيث يستخدم التوسع المتأخر بدلاً من التوسع المبكر.

```
private fname := "var1"
private var1 := 1
private var2 := 2
b := { | | &(fname) }
? eval(b)                       // output: 1
fname := "var2"
? eval(b)                       // output: 2
```

وتجدر الإشارة إلى أنه إذا استخدمت طريقة "التوسع المتأخر" فقد تواجهك عقبة بسيطة في الأداء وذلك لأن توسيع الماكرو هو عملية بطيئة نسبياً ، وقد يضطر البرنامج إلى تقييم كتلة الشيفرة المطلوبة عشرات أو مئات المرات ، لذا يحسن الانتباه إلى هذا الأمر قبل استخدام هذه الطريقة لأنها تعيق عملك.

تمرير متغيرات محلية من خلال كتل الشيفرة

لعلك تذكر أن مجال المتغير المحلي هو الإجراء أو الوظيفة التي سيتم إعلانه فيها. إلا أن هناك طريقة يمكن تمرير متغير محلي إلى وظيفة أخرى ، وتتطلب هذه الطريقة استخدام كتل الشيفرة ، كما يلي:

```
function main
local block := { | | x }
```

```
local  x  :=  500
test1(block)
return  nil

function test1(b)
?  eval(b)                    //  output: 500
return nil
```

فعندما يتم إنشاء كتلة شيفرة وتجميعها باستخدام وظيفة MAIN( ) ستحتوي على إشارة إلى x وهو متغير محلي بالنسبة للوظيفة MAIN() ، إلا أنه عندما يتم تمرير الكتلة على أنها متغير إلى الوظيفة TEST1() ، حيث يتم تقييمها هناك ، وبالتالي فستصبح قيمة x موجودة فعلياً هناك في الوظيفة التي تم تمريرها إليها TEST1().

وتجدر الإشارة إلى أنه لن يختلف الأمر تماماً سواءً أتم تأسيس قيمة المتغير x قبل كتلة الشيفرة أو بعدها ، والمهم في الأمران أن يوجد المتغير x قبل عملية تقييم كتلة الشيفرة التي تشير إليه.

ولننظر إلى المثال السابق مرة أخرى ، ولكن بإضافة إعلان المتغير المحلي x في الوظيفة على المستوى الأدنى:

```
function main
local  block := {  |  |  x  }
local  x  :=  500
test1(block)
return  nil

function test1(b)
local  x  :=  100
?  eval(b)                    //  output: still 500
return nil
```

قد يتراءى لك من النظرة الأولى أن تقييم كتلة الشيفرة قد يعطيك قيمة المتغير x محلياً بالنسبة للوظيفة TEST1( ) ، وليس هذا صحيحاً. فكما ذكرنا آنفاً فإن كتلة الشيفرة هي كتلة مجمعة ، إلا أن الاسم الذي عينته لها يمثل عنوان الذاكرة. وهكذا ، فإذا مررت

كتلة شيفرة على أنها متغير لوظيفة أخرى ، فإنك تمرر عنوان الذاكرة الذي تبدأ منه الكتلة المجمعة فقط.

وإن أهم شيء يجب أخذه بعين الاعتبار عند تقييم كتلة شيفرة هو "المحتوى الذي تم تأسيسه بموجبها ، وليس المحتوى الذي يتم تقييمها بناءً عليه".

أثر كتلة الشيفرة

لقد تعلمنا أثناء دراستنا لبرنامج اكتشاف الأخطاء البرمجية وتصحيحها شيئاً عن خيار "أثر كتلة الشيفرة code block tracs" ونحذرك هنا أيضا ، بأنه يجب ألا توقف عمل هذا الخيار على الإطلاق!!. وإذا لم تع هذه النصيحة فإنك إذا بدأت بتنفيذ برنامجك وبدأت بتقييم كتل الشيفرة فإن برنامج اكتشاف الأخطاء وتصحيحها لن يقفز إلى السطر الذي تم تحديد البرنامج عنده. وكما رأينا في المثال السابق فإنه يتم تقييم كتل الشيفرة دائماً في المحتوى الذي تم تأسيسها فيه ، بغض النظر عن مكانها في البرنامج. لذا، فإذا أردت إعادة تنفيذ المثال السابق بحيث يكون أثر كتلة الشيفرة موقفاً عن العمل، فإنك لن تستطيع معرفة سبب إخراج ٥٠٠ بدلا من ١٠٠؟ ونعتقد أن هذا كاف.

المحليات المنفصلة Detached locals

عرفنا كيف يمكننا تمرير متغيرات محلية إلى وظائف دنيا باستخدام كتل شيفرة. وإن كليبر 5.x ، يمكنك أيضاً من تمرير متغيرات محلية من وظائف دنيا إلى وظائف عليا بطريقة مشابهة. ولابد هنا من إنشاء كتلة شيفرة في وظيفة على المستوى الأدنى تشير إلى متغير محلي ، ثم تمرر تلك الكتلة إلى وظيفة أعلى. ويبين الجزء التالي من البرنامج آلية عمل هذه الطريقة:

```
function  test
local myblock
```

```
myblock := clipver( )
? eval(myblock)
return nil

static function clipver
local xx := "CA-Clipper 5.x"
return { | | xx }
```

تعرف هذه المتغيرات المحلية بأنها "محلية منفصلة" إذ أنها تبقى حتى بعد انتهاء عمل الوظيفة التي أنشأتها ، بل إن هذه المتغيرات ستستمر في البقاء طيلة بقاء كتلة الشيفرة التي تشير إليها. وقد تعجب قائلاً: " لماذا أرغب في استخدام هذه المتغيرات؟" ولاشك أنك محق في هذا السؤال. سنبني الآن مصفوفة من كتل شيفرة ، كل من هذه الكتل تشير إلى عنصر في مصفوفة أخرى. وقد تبدو هذه الكتلة للوهلة الأولى سليمة تماماً:

```
function test
local a := { "One", "Two", "Three", "Four", "Five" }, b := { }
local x
for x := 1 to 5
   aadd(b, { | | a[x] } )
next
? eval(b[1])
return nil
```

إلا أنك فور ماتحاول تقييم أول كتلة شيفرة في المصفوفة B سيتحطم البرنامج مصدراً رسالة تقول:"Bound Array Access" (تعامل مع مصفوفة مشروطة). فلماذا يحدث هذا؟ أولا: يجب أن تتذكر أنه كلما كان لديك حلقة FOR...NEXT وسيكون عداد الحلقة مساوياً للحد الأقصى للحلقة زائداً قيمة خطوة الحلقة. وهنا ستكون قيمة x في هذا المثال هي 6 ، وذلك بمنتهى البساطة.

ثم إن كتلة الشيفرة التي تشير إلى X لن تحل ، أو تبرمج قيمة X الحالية عندما تؤسسها. لذلك ، فإنك كلما تقييم كتلة الشيفرة ستنظر إلى القيمة الحالية للمتغير X ، ولعلك الآن تدرك حقيقة ماجرى. إنك تحاول إرجاع قيمة [6] للمصفوفة A. وبما أن تلك المصفوفة تحتوي على ٥ عناصر فإنك لن تحصل على نتيجة لذاك التقدير.

وإليك فيما يلي حل يستغل مبدأ " المحليات المنفصلة" detached locals:

```
function test
local a := { "One" , "Two", "Three", "Four", "Five" }
local b := { }
local x
for x := 1 to 5
    aadd(b, makeblock(a, x) )
next
? eval(b[1])
return nil

static function makeblock(array, ele)
return { | | array[ele] }
```

تقوم وظيفة MakeBlock( ) ببرمجة قيمة X في كتلة الشيفرة بشكل فعال ، وتجدر الإشارة إلى أنه يجب تمرير المصفوفة إلى الوظيفة ذاتها ولن يكفي أن تمرر الوظيفة إلى عنصر المصفوفة فقط.

إن المحليات المنفصلة detached locals مفيدة جداً عندما تريد إنشاء عنصر Tbrowse لتتمكن من الاطلاع على مصفوفة تحتوي على مصفوفات متداخلة حيث أنك لاتعرف طول المصفوفات المتداخلة ، وهي عملية أيضاً في التطبيقات التي تستخدم شاشات GET التي تستخدم البيانات.

ولعل "المحليات المنفصلة" هي أصعب شيء يمكن شرحه وتفسيره ذهنياً من موضوعات كليبر. ويلاحظ في لغة سي أن كلاً من المتغيرات المحلية والساكنة توضع في مجموعات ، إلا أنه نظراً لأن الوظيفة التي أنشأت المتغيرات المحلية المنفصلة غير موجودة في المجموعة ، فلابد أن هناك شيئاً من الغلط في البرنامج.

ولعل أفضل وسيلة لفهم المتغيرات المنفصلة هي أن نعرف أننا عندما نرجع كتلة شيفرة من وظيفة ما ، فإن كليبر سيقوم باختبار تلك الكتلة ليعرف ما إذا كانت هناك أية إشارة إلى أي شيء محلي Local في تلك الوظيفة ، فإذا كان ذلك كذلك ، فإنه سيتم إنشاء مصفوفة تحتوي على ذلك المتغير (أو المتغيرات). وإن عنوان ذاكرة

المصفوفة، مع ما ينشأ عنها من مصفوفة فرعية لكل متغير محلي يتم حفظها لمراجعتها مستقبلاً. وعندما يتم تقييم كتلة الشيفرة لاحقاً ، فإن تسلسل البرنامج سيقفز إلى عنوان الذاكرة الذي تم حفظ المصفوفة فيه لاستعادة القيمة المحلية.

وقد بينا أن عملية "المشاهدة أو البحث" هذه ستكون أبطأ قليلاً من استدعاء القيم من المكدس stack مباشرةً ، وذلك لأنه سيتم استدعاؤها عن طريق المصفوفة. إلا أنه لدى استخدام المحليات المنفصلة ، التي تنجيك من هذا المأزق للمرة الأولى ، فإنني أعتقد أنه يمكننا جميعاً التعامل مع هذا البطء النسبي الذي يمكن تجاوزه فيما يتعلق بالاداء إذا قدرنا النتيجة التي سنحصل عليها.

تحذير الانفصال المتأخر للمحليات المنفصلة

تقدمت هيئة تطوير كليبر الألمانية في ديسمبر، كانون أول ١٩٩٢، بتقرير يقول أن: "كل ماتراه عند إنشاء البرنامج سيبقى حياً (مثال: العامة public والخاصة private والساكنة static على مدى الملف) ، وليس المحلية فقط. وقد يكون هذا مزعجاً ، إذ أنه لو تم إعداد نسخة من هذه العناصر الأخرى ، فإنك لن تستطيع أن تفعل أي شيء بها. كما أن الآثار الجانبية الأخرى وهو أن الاستخدام الإضافي للذاكرة غالباً ما ينتج عنه الخطء المخيف الداخلي (وغير الممكن الاسترجاع) وهو الخطأ ٥٣٣٣. لذلك يستحسن عند استخدام المحليات المنفصلة التأكد من بقاء أقل عدد من العناصر مرئية وإذا لم تستخدم الإعلانات العامة والخاصة فسيلغي هذا إمكانية حدوث معظم المشاكل.

الوظائف التي تتطلب كتل شيفرة

الوظيفة:

EVAL (<block>,[<arg list>]

لا شك أن تدرك أن هذه الوظيفة تقيّم كتلة البرنامج الذي ترسله إليها على شكل متغير "كتلة". وإن المتغير الاختياري <arg list> يفصل بفاصلة حتى يتم إرساله إلى كتلة الشيفرة عندما تريد تقييمها.

القيمة الراجعة: كما ذكرنا سابقاً، إن وظيفة ( )EVAL ترجع القيمة لآخر (في أقصى اليمين) تعبير ضمن تلك الكتلة.

الوظيفة:

AEVAL(<arry>, <block>, [<start>, [<count>)

إن وظيفة ( )AEVAL تشبه وظيفة ( )EVAL إلا أنها مصممة خصيصاً لتعمل مع المصفوفات. فهي تقيم كتلة البرنامج (المحددة بموجب متغير <block>) لكل عنصر في المصفوفة (محدد بموجب متغير <array>). ويمكنك أن تختار استخدام عنصر <start>، وعدداً من العناصر (<count>) لمعالجتها. وإذا لم تستخدم هذه المتغيرات الاختيارية فستبدأ وظيفة ( )AEVAL بأول عنصر في المصفوفة، وتعالجها جميعها.

إن الوظيفة التالية ( )AEVAL هي عامل جيد إذ أنها تقرر كلاً من الحد الأعلى، والحد الأدنى، والمجموع لكافة العناصر في المصفوفة MyArry:

```
local myarray := { 75, 100, 2, 200, .25, -25, 40, 52 } , ;
      nmax, nmin, nsum := 0
nmax := nmin  := myarray[1]
aeval(myarray,  {  |  a  |  nmax := max(nmax, a) , nmin := min(nmin, a), ;
                        nsum += a }  )
? "Maximum Value:",  nmax           // 200
? "Minimum  Value:",  nmin          // -25
? "Total amount:" ,    nsum         // 444.25
```

ترسل وظيفة ( )AEVAL متغيرين بشكل آلي إلى كتلة الشيفرة <value> و <number>، حيث تكون <value> هي قيمة عنصر المصفوفة الذي يتم معالجته، ويكون <number> رقم عنصر المصفوفة التي تتم معالجتها. ولقد رأيت كيف تستخدم

وظيفة <value>، ولكن ترى لماذا نزعج أنفسنا باستخدام وظيفة <number>؟ لنفترض أنك تريد زيادة كل عنصر في المصفوفة MyArray. فمن المحتمل أن كتلة برنامجك على النحو التالي:

```
aeval(myarray, { | a | a++ } )
aeval(myarray, { | a | qout(a)  } )
```

والمفاجأة أن هذه الكتلة لن تقوم بأي شيء تجاه عناصر المصفوفة لأنها قد أرسلت باستخدام القيمة (وليس الإشارة المرجعية) إلى كتلة الشيفرة. إن الإرسال باستخدام القيمة يعني أن كتلة الشيفرة ستعمل نسخة من عنصر المصفوفة وأن أي بلورة تتم داخل كتلة الشيفرة ستتم على النسخة، وليس على العنصر الحقيقي. إذن، لنحاول إجراء العملية من جديد باستخدام المتغير <number>:

```
aeval(myarray, { | a, b  | myarray[b]++  } )
aeval(myarray, { | a | qout(a)  } )
```

القيمة الراجعة: سترجع ( )AEVAL إشارة إلى المصفوفة التي طلبت معالجتها.

الوظيفة:

```
DBEVAL(<block>, [<for>], [<while>], [<next>], [<record>], [<rest>])
```

تشبه هذه الوظيفة وظيفة ( )AEVAL إلا أنها تتعامل مع قواعد البيانات بدلاً من المصفوفات، كما أنها تقدم سيطرة أكبر بكثير منها إذ أنها تستخدم كلاً من شروط: FOR, WHILE, NEXT, RECORD و REST. وإذا نظرت إلى ملف الترويسة STD.CH ستلاحظ أن كلاً من أوامر: REPIACE, RECALL, DEIETE, والتي تم استبدالها بـ: AVERAGE, SUM, COUNT هي معالجة مسبقاً في استدعاءات ( )DSEVAL. فإذا أردت مثلاً أن تجمع حقل الباقي لكافة السجلات الموجودة في قاعدة بياناتك، فسيقوم الأمر التالي بتنفيذ ذلك:

```
ntotal := 0
DBEval( { |  | ntotal += balance} )
```

إن أحد الأمور السلبية لهذه الأوامر COUNT, SUM وغيرها هو أنه لا يريك أي رد أو تغذية راجعة عما قام به من عمل. إلا أنه يمكنك تعديل هذا بسهولة بحيث يمكنك الحصول على تغذية راجعة على النحو التالي باستخدام هذا الأمر. فتطبع كتلة الشيفرة التالية رقم السجل الحالي والإجمالي أثناء القيام بإجراء حلقة داخل قاعدة البيانات:

```
local nSum  :=  0
use  customer  new  alias cust
DBEval(  {  |   |  setpos(0,  0),  ;
                   dispout(recno( )  ),  ;
                   dispout(nSum  +=  cust->balance)  }   )
?  "Total:   ",    ntotal
```

لاحظ استخدام أمري SETPOS( ) ووظيفة DISPOUT( )، فالأول تحدد موقع المؤشر عند سطر وعمود محددين، وأما DISPOUT فتعرض القيمة على الشاشة. وكما ذكرنا سابقاً، فإنه لا يمكنك استخدام SAY..@ في هذه الحالة إذ أن المعالج الأولي لا يمكنه ترجمته بشكل مناسب وصحيح. كما يمكن استخدام وظيفة DBEVAL( ) لمتابعة أعلى باقٍ ولمتابعة الإجمالي أيضاً، وذلك على النحو التالي:

```
use customer new alias  cust
ntotal  :=  nmax  :=  0
DBEval(  {  |   |  ntotal   +=  cust->balance,  nmax  :=;
                 max(nmax,cust->balance)  }  )
?  "Total:      ",  ntotal
?  "Maximum:",  nmax
```

متغيرات DBEVAL( )

إن المتغير <block> هو كتلة الشيفرة اللازمة تقييمها مكل سجل من سجلات قاعدة البيانات. وهناك عدد من المتغيرات الاختيارية نذكر منها:

<For> و <while> فهما متغيران لكتل الشيفرة يطابقان كلاً من FOR و WHILE مباشرة. فإذا استخدمت أياً منهما، أو استخدمتهما معاً فإن وظيفة DBEVAL( ) ستعالج السجلات إلى تصبح أن القيمة المرجعة من كتل الشيفرة هي (غير حقيقية)

(.F.) ويبين هذا الجزء التالي من البرنامج المثال السابق، ويتابع الإجمالي، والباقي الأكبر لكافة السجلات في ولاية محددة على حين أن مؤشر السجل هو أقل من ٢٠٠:

```
use customer new  alias cust
ntotal  := nmax  := 0
DBEval( { |   |  ntotal  += cust->balance,  nmax  :=  max(nmax, ;
                                  cust->balance)  } ,  ;
        { |   |  cust->state == "CA"} ,  { |   | recno( )  <  200} )
? "Total :      " ,  ntotal
? "Maximum: " , nmax
```

متغيرا <next> و <record> رقميان. فمتغير <next> يحدد عدد السجلات الواجب معالجتها بدءاً من السجل الحالي. والمتغير <record> يحدد رقم السجل المراد معالجته. وإذا نظرنا ثانية على المثال السابق، إلا أننا سنعالج في هذه المرة السجلات المئة التالية للعملاء في ولاية أخرى مثلاً:

```
use customer  new  alias  cust
ntotal  :=  nmax  := 0
DBEval( { |    |   ntotal +=  cust->balance, nmax :=  max(nmax, ;
                                     cust->balance)  } ,  ;
        { |    |  cust->state  == "OR" } , , 100 )
? "Total :      " ,  ntotal
? "Maximum: " ,  nmax
```

أما المتغير <rest> فهو متغير منطقي يقرر ما إذا كانت عملية ( )DBEVAL ستبدأ من السجل الحالي إلى نهاية الملف، أو السجلات جميعها. فإذا أرسلت القيمة المنطقية "حقيقي" (.T.) فسيفترض أنك تريد الأول أي ابدأ من السجل الحالي، وإذا أرسلت القيمة المنطقية "غير حقيقي" (.F.) أو إذا تجاهلت هذا المتغير تماماً، فإن هذه الوظيفة ستعالج كافة السجلات.

القيمة الراجعة: سترجع وظيفة ( )DBEVALE قيمة الصفر.

الوظيفة:

```
ASCAN(<array>, <value>, [<state>], [<count>])
```

تشبه هذه الوظيفة عملها ذاته في الإصدار Summer'87 من كليبر فهي تمسح مصفوفة للبحث عن قيمة محددة، إلا أن الفارق الكبير بينهما هو أنك هنا تستطيع باستخدام

هذه الوظيفة إرسال كتلة شيفرة كمتغير للقيمة <value>. ويجب أن تكتب كتلة الشيفرة هذه بحيث تقبل متغيراً واحداً وبحيث ترجع قيمة منطقية. ويمثل هذا المتغير كل عنصر من عناصر المصفوفة. وستقوم هذه الوظيفة بتقييم كتلة الشيفرة لكل عنصر من العناصر في المصفوفة، وإذا أرجعت كتلة الشيفرة قيمة منطقية حقيقية (.T.) ستوقف هذه الوظيفة فوراً وسترجع رقم العنصر الحالي. وأما إذا أرجعت الكتلة قيمة "غير حقيقية" (.F.) فستعيد الوظيفة المعالجة بدءاً من عنصر المصفوفة التالية. لاحظ أن كل ما يجب عليك القيام به هو كتابة كتلة الشيفرة بشكل مناسب، وسيقوم كليبر بتقييمها لك. وقد تسأل: لماذا نفعل مثل هذا؟ إن إحدى الحالات المفيدة، هي أن تجبر وظيفة ASCAN( ) لتتحرى وجود الحالة التي لا يمكن أن يتحسس الكمبيوتر بها. فإن هذه الوظيفة هي تتحرى وجود الحالات التي يتحسس بها الكمبيوتر بشكل مفترض، إلا أن استخدام كتلة شيفرة توضع في مكان مناسب يمكنه إلغاء هذا التحسس المفترض كما يبينه المثال التالي:

```
local myarray := { "Ahmad", "Turki", "Yaser", "Mohammed" }
local searchval := "Mary"
// -------- first search will fail
? ascan(myarray, searchval)
//---------second search (with code block) will succeed
? ascan(myarray, { | name | upper(name) == upper(searchval) } )
return nil
```

إذا أردت مسح مصفوفة تحتوي على عدة مصفوفات داخلها فإنه لن يمكنك هذا دون استخدام كتلة الشيفرة. مثلاً: إذا مسح الدليل الأساسي/الجذري للقرص الصلب :C بحثاً عن ملف AUTOEXEX.BAT فإن أفضل طريقة للقيام بهذا باستخدام كليبر هو استخدام وظيفة DIRCTORY( ) والتي ترجع مصفوفة تحتوي على مصفوفات بداخلها لكل ملف يطابق ما حددت له. ويكون تركيب هذه المصفوفات الداخلية على النحو التالي:

عنصر المصفوفة	المعلومات	ثابت البيان في ملف DIRECTRY.CH
1	file name	F_NAME
2	file size	F_SIZE
3	file date	F_DATE
4	file time	F_TIME
5	file attribute	F_ATTR

لذا، سنكتب كتلة الشيفرة الخاصة بنا بحيث تفحص العنصر الأول فقط من كل مصفوفة داخلية:

```
local files_ := directory("C:\*.*")
if ascan(files_ , { | f | f[F_name] == "AUTOEXEC.BAT" } ) > 0
   ? "found it"
endif
return nil
```

الوظيفة:

```
ASORT(<array>, [<start>] , [<count>] , [<block>]
```

تشبه هذه الوظيفة مماثلتها في إصدار Summer'87، وأما المتغيران الاختياريان <start> و <count> هما كما كانا في وظيفة AEVAL( ). إلا أنك باستخدام هذه الوظيفة (الفرز) فإن كتل الشيفرة تمكنك من تغيير شكل الأشياء بشكل رهيب، ويمكنك أن تقوم بعمليات فرز متنوعة الأشكال والأغراض والأهداف. فيمكن أن تضع كافة العناصر التي تحتوي على مصوفات فرعية في أعلى المصفوفة. ويمكنك الفرز بطريقة تنازلية، أو الفرز الألفبائي، بناء على الحرف الأخير من الكلمة (!) وغير ذلك من الأمور الطريفة. وستصدر الوظيفة في كل مرة يتم فيها تقييم كتلة الشيفرة الخاصة بك باستخدام الوظيفة ASORT( ) عناصر مصفوفة إلى الكتلة. ويتوقع من الكتلة عندئذ مقارنة هذه العناصر بطريقة تحددها له أنت، ويرجع قيمة منطقية إما "حقيقي" (.T.) إذا كانت العناصر بالترتيب الصحيح، أو قيمة "غير حقيقي" (.F.) إذا لم تكن بالترتيب الصحيح. وإليك مثال على الفرز التنازلي:

```
local myarray := { "Fahad" , "Basma" , "Tameer" , "Dalal" }
asort(myarray , , , { | x, y | x > y } )
aeval(myarray, { | a | qout(a) } ) // to show it worked!
```

وكما هي الحال باستخدام وظيفة ASCAN( ) إذا أردت معالجة مصفوفة تحتوي على عدد من المصفوفات داخلها، فلا بد من الاعتماد على كتلة الشيفرة. ويمكن أن يكون المثال الجيد على هذه الحالة فرز ملف معلومات بناء على اسم الملف. ففي إصدار Summer'87 كان الاعتماد على وظيفة DIR( )، والذي سيتم تنسيقه في المستقبل القريب، والذي يتطلب تأسيس مصفوفة لكل من المعلومات التي تود الحصول عليها.

```
* sort a directory listing by filename
* first in Summer '87
private files_[adir("*.*") ]
adir("*.*" , files_)
asort(files_)

* then in CA-Clipper 5.x
local files_ := directory("*.*")
asort(files_ , , , { | x, y | x[F_NAME] < y[F_NAME] } )
```

والآن، إذا أردنا فرز الأدلة بناء على التاريخ:

```
* Summer '87
private files_[adir("*.*") ] , dates_[adir("*.*")]
adir("*.*" , files_, ' ' , dates_)
asort(dates_)

* CA-Clipper 5.x
local files_ := directory("*.*")
asort(files_ , , , { | x, y | x[F_DATE] < y[F_date] } )
```

وستلاحظ أن شيفرة Summer'87 تصبح مثقلة كلما أضفت مصفوفات أخرى، وكذلك عند الفرز بناء على التاريخ فإن مصفوفات الملفات (التي تحتوي على أسماء الملفات) ستترك كما هي دون تغيير، وبهذا تخسر كل الجهد الذي بذلته ولا تستفيد

شيئاً. وذلك لأنه لدى استخدام وظيفة ADIR( ) فإن جزءً من معلومات الملف قد تم تخزينه في مصفوفة مستقلة، وبهذا فإنه غير مرتبط منطقياً بالأجزاء الأخرى. وبالمقابل، فإن استخدام وظيفة DIRECCORY( ) ينتج عنه مصفوفة متداخلة لكل ملف. وهذا يعني أنه يمكنك الفرز بناء على المتغيرات دون الاهتمام بترك أي من معلومات الملف دون فرز. والآن، لنفرز الدليل طبقاً للتاريخ والاسم:

```
* Summer '87
* لقد عجزت!
* CA-Clipper 5.x
local files_ := directory("*.*")
asort(files_ , , , { | x, y | if( x[F_DATE] == y[F_DATE] , ;
                                  x[F_NAME] < y[F_NAME] , ;
                                  x[F_DATE] < y[F_DATE] ) } )
aeval(files_ , { | a | qout(padr(a[F_NAME] , 14) , a[F_DATE] } )
```

(لاحظ استخدام وظيفة PADR( ) لتضمن أن أسماء الملفات مرصوصة تحت بعضها بشكل سليم).

ويمكنك أن تحدد بسهولة إذا كانت التواريخ هي ذاتها باستخدام (x[F_DATE] = = y[F_DATE]) فإذا كانت كذلك فستتم مقارنة أسماء الملفات (x[F_NAME] < y[F_NAME]). وإذا لم تكن كذلك، فتقارن تواريخ الملفات ([F_DATE] < y[F_DATE]). وهذا مثال صغير فقط على بعض الاشياء التي يمكنك القيام بها الآن باستخدام كليبر والتي لم تكن تستطيع القيام بها في الإصدارات السابقة.

ومثال آخر على مرونة وظيفة ASORT( ) هو السؤال التالي: أراد أحدهم فرز مصفوفة تحتوي على سلاسل حرفية، إلا أن هناك سلاسل فارغة فيها وضعت في أسفلها. والجواب على ذلك، هو كيفية تركيب كتلة الشيفرة التي ترسلها إلى وظيفة ASORT( ). يمكننا تصفية السلاسل الفارغة بإضافة اختيار إضافي لوظيفة EMPTY( ) في كتلة الشيفرة، وسيتم في هذه الحالة معالجة السلاسل الممتلئة فقط طبقاً للمقارنة التصاعدية العادية. فإذا كانت أحد السلاسل فارغة فسيكون راجع كتلة

الشيفرة هو "حقيقي" (.T.) إذا كانت السلسة الأولى فارغة. ولن يسمح هذا بوجود سلاسل فارغة بين السلاسل التي تتم معالجتها. ويبين المثال التالي كيفية عمل هذا المنطق:

```
function  main
local  a := {  "TEST" , "JONES", "CRIMSON", "MAGENTA" ,  ;
               " " , "OREGON" , "SYDNEY" , "HAWAII" ,  ;
               " " , "SINGAPORE",  " ",  "LONDON"  }
asort(a,  ,  , { |  x,  y  |  ;
                 if(!  empty(x)  .and.  !  empty(y) ,  ;
                   x  <  y ,  !  empty(x)  )  }  )
// display  all  elements  to  show  that  it  worked
aeval(a,  {  |  string,  count  |  qout(count,  string)  }  )
inkey(0)
return nil
```

الوظيفة:

SETKEY(<key> , [<block>])

إذا كنت قد استخدمت أمر SET KEY في برنامج Summer'87 لتأسيس مفتاح مباشر (Hot Key)، فلعلك تذكر الإحباطات التي حدثت أثناء ذلك. فمثلاً: إذا أردت إيقاف عمل كافة "المفاتيح المباشرة" إذا كان المستخدم يعمل في إجزاء يستخدمها فيه، فقد تطلب هذا العمل منك عملية برمجة معقدة ومتعبة. وأما في كليبر الجديد، فهذه منطقة أخرى يعطيك البرنامج فيها إمكانية عالية على التحكم والسيطرة لم يسبق لها مثيل. وإنك عندما تحدد "مفتاحَ مباشراً" باستخدام أمر SET KEY فإنك تربط كتلة برمجة بالضغط على ذلك المفتاح باستخدام وظيفة ( )SETKEY، التي تمكنك من استلام قيمة أي ( )INKEY لتقرر ما إذا كانت هناك كتلة برمجة مرتبطة بها أم لا. وكما هي الحال في مختلف وظائف ( )SET فهي تسمح لك أيضاً بتغيير التجهيزات الحالية: أي: ربط كتلة برمجة بأي مفتاح آخر.

إن متغير <key> هو متغير رقمي يقابل وظيفة ( )INKEY للمفتاح الذي سيتم الضغط عليه. وأما المتغير الاختياري <block> فهو كتلة الشيفرة التي سيتم تقييمها إذا كان المفتاح المضغوط عليه هو خلال عملية انتظار.

وتتضمن حالة الانتظار كلاً من الوظائف التالية: ()ACHOICE و ()DBEDIT و ()MEMOEDIT و ACCEPT و INPUT و READ و WAIT وأخيراً .MENU TO

وأما وظيفة ( )SETKEY فإما أن ترجع كتلة الشيفرة إذا كانت هناك واحدة قد تم ربطها بمتغير <key> أو لا شيء. وإذا أرسلت متغير <block> فإنها سترتبط كتلة الشيفرة بمتغير <key>.

أمر SETKEY

قبل الاطلاع على أمثلة ( )SETKEY، لنطلع معاً على أمر SET KEY وكيف يتم التعامل معها من خلال كليبر.

```
set key  28  to helpdev
```

والتي ستتم ترجمتها من قبل المعالج الأولي على النحو التالي:

```
SetKey( 28, { | p, 1, v | helpdev(p, 1, v ) } )
```

وتقابل كل من المتغيرات P , L , V وظيفة ( )PROCNAME (اسم الإجراء) و ( )PROCLINE (رقم سطر شيفرة المصدر) و ( )READVAR (اسم المتغير) والتي سيتم إرسالها بشكل آلي إلى كتلة الشيفرة عند تقييمها. إلا أنه يمكنك حذف هذه المناقشات في إعلان كتلة برمجتك إذا لم تكن ستستخدمها هناك. وكذلك في الوقت ذاته، لك مطلق الحرية لإرسال متغيرات مختلفة تماماً عن الوظيفة الموجودة في قائمة تعابير كتلة الشيفرة. (وسنبين هذا بعد قليل).

إنك كلما وصلت إلى حالة أنتظار في كليبر فإن ضغطة المفتاح سيتم تقييمها بهذه الطريقة تقريباً لتقرر ما إذا كان هناك "مفتاح مباشر" مرتبط بهذا الإجراء أم لا.

```
keypress := inkey(0)
if  setkey(keypress)  != NIL
   eval(setkey(keypress) )
endif
```

تنظيم أفضل باستخدام وظيفة SETKEY( )

نبين فيما يلي كيف أن استخدام هذه الوظيفة سيحقق اختلافاً واضحاً بين إيجاد حل أو خلافه: لنفترض أنك خلال إجراء تحديد "مفتاح مباشر" أردت أن تربط –بشكل مؤقت– تعريف مفتاح مباشر إلى المفتاح المباشر F10، إلا أنك قد حددت عدداً من الإجراءات المختلفة باستخدام F10 خلال برنامجك. ففي برنامج Summer'87 أحدث هذا مشكلة كبيرة لأنك لم تستطع تحديد الإجراء المحدد الذي تم ربطه بمفتاح الأوامر المباشر F10. لذلك لم تكن قادراً على تغييره وإعادة تجهيزه بشكل مناسب. ولم يعد هذا مشكلة في كليبر باستخدام وظيفة SETKEY( ) حسب المثال التالي:

```
#include  "inkey.ch"                // for  INKEY( )  constants (الثوابت)
function  test(p,  1,  v)
local  old_f10  :=  setkey(K_F10,  {  | p,1,v|  HelpIndex(p,1,v)  }  )
*  main  code  goes  here
setkey(K_f10, old_f10)
return nil                                  // restore  F10  hot  key
```

المساعدة الحساسة ووظيفة SETKEY( )

إن كلاً من مرونة كتلة الشيفرة ووظيفة SETKEY( ) تمكنك من استخدام مساعدة تتحسس حسب المحتوى بشكل دقيق وأكبر من خلال برامج تعدها باستخدام كليبر.

لنبدأ بتحديد متغير رقم السطر الذي يتم إرساله مباشرة إلى وظائف المفاتيح المباشرة. فكما ذكرنا سابقاً فإن "رقم السطر" متطاير جداً بحيث لا يمكن استخدامه بشكل مفيد. لذا، يجب أن تصوغ وظيفة HELP( ) بحيث لا تقبل سوى كل من الإجراء واسم المتغير. يمكن بتجهيز مفتاح الأوامر المباشرة F1 (أو أي مفتاح آخر تختاره) لإرسال هذين المتغيرين فقط إلى وظيفة HELP( ). لاحظ أنه يجب أن تقبل المتغيرات الثلاثة جميعها بحيث يمكنها التوصل إلى اسم المتغير (V).

```
#include "inkey.ch"              // for inkey( ) constants

function main
local old_f1 := setkey(K_F1, { | p, 1, v | Help(p, v) } )
//
// body of function
//
setkey(K_f1, old_f1)
return nil

function help(p, v)
if help->( dbseek( padr(p + v, 20) ) )
        // display help screen
else
   alert("No help available")
endif
return nil
```

لاحظ استخدام وظيفة PADR( ) لدى البحث من شاشة مساعدة لفترة الانتظار هذه، ويعتبر الإجراء واسم المتغير يمكن أن يكون لكل منهما طولاً أعظمياً يبلغ عشرة رموز.

ولنفترض الآن أن لديك عدة استدعاءات لوظيفة ACHOICE( )، فإذا ضغطت على مفتاح مباشر أثناء وظيفة ACHOICE( ) فإن كليبر سيعطيك دائماً المعلومات ذاتها بالنسبة لفترة الانتظار. ولن يكون هذا مساعداً جداً عندما تريد الحصول على شاشات مساعدة مختلفة لكل مرة تستدعي فيها وظيفة ACHOICE( ) ويبين لك المثال التالي حلّ مثل هذه المشكلة:

```
#include "inkey.ch"              // for inkey( ) constants

function main
```

```
local old_f1 := setkey(K_F1, { | | Help("ACHICE1") } )
achoice(...)
setkey(K_f1, old_f1)
return nil
function help(p, v)
```

لاحظ أن كتلة الشيفرة هنا لم تكتب لتقبل متغيرات p , L , V، وفي هذه الحالة لا ضرورة لذلك لأننا نتجاوزها باستخدام برمجة محددة تطلب شاشات المساعدة.

استخدام وظيفة INKEY( ) كحالة انتظار

إن هذه الوظيفة ليست مضمونة كما هي الحال في برنامج Summer'87، ولكن كما قد أوضحنا لك قبل قليل إن وظيفة SETKEY( ) تسهل إنشاء وظيفة حالة انتظار خاصة بك. وإن الوظيفة التالية، وهي: MyInKey( ) تستخدم وظيفة SETKEY( ) لتختبر ضغطة المفتاح لمفتاح مباشر. وبدلاً من استدعاء وظيفة INKEY( ) في برنامجك، يمكنك استدعاء وظيفة MyInKey( ) بدلاً منها.

```
function myinkey
local key := inkey(0)
local block := setkey(key)
if block != NIL                    // there is a code block for this key
   eval(block, procname(1), procline(1), 'myinkey')
endif
return key
```

إذا أرجعت وظيفة INKEY( ) كتلة برمجة فسيتم تقييمها. لاحظ أننا نقيمها بدلاً من إرسالها في الإجراء الحالي ورقم السطر، بل نرسل إليها المعلومات التي هي على مستوى أبعد واحداً من آخر كومة التنشيط، وإلا فإنه إجراء المفتاح المباشر سيعتقد دوماً أنه آت من وظيفة MyInKey( ) وسيحدث هذا الأمر كثيراً من الفوضى والاضطراب ويجبرك على الحصول على شاشة المساعدة ذاتها لكل حالة انتظار تستدعيها وظيفة MyInKey( ).

تمييع وظيفة INKEY( ) لحالة الانتظار

بعد أن اطلعنا على أساسيات هذه الوظيفة، يمكننا أن نتوسع في الأمور التالية:

- أسماء متغيرات مختلفة لحالات انتظار مختلفة.
- حادثة خلفية مستمرة اختيارية.
- فترة محددة مستقطعة وحادثة اختياريتان.
- أمر يحدده المستخدم لتسهيل القراءة.

أسماء متغيرات مختلفة لحالات انتظار مختلفة

إن المتغير الثالث الذي يرسل إجراءات المفاتيح المباشرة كما ذكرنا آنفاً هو اسم المتغير الذي سيقرأ. وفي الوظيفة التالية "MYINKEY( )" سيعتبر اسماً وهمياً للمتغير. إلا أن هذا يعني أنك إذا استدعيت وظيفة MYINKEY( ) أكثر من مرة في الإجراء ذاته فإنك ستحصل على شاشة المساعدة ذاتها لكل حالة من حالات الانتظار لأن كلاً من متغيري الاجراء واسم المتغير سيكونان ذاتهما. وإن من السهولة بمكان تعديل وظيفة MyInKey( ) لتقبل اسم المتغير كمتغير، وهذا ما سنفعله بالضبط.

حادثة خلفية مستمرة اختيارية

لننظر مرة ثانية على أهم سطرين في وظيفة MyInKey( ):

```
local key := inkey(0)
```

لا يوجد ما يمنعنا على الإطلاق من إعادة كتابة هذا على شكل حلقة باستخدام DO WHILE :

```
do while ( key := inkey( ) ) == 0
enddo
```

وستستمر هذه الحلقة بسحب ما يوجد في الذاكرة المؤقتة للوحة المفاتيح. وعند اكتشاف ضغطة على لوحة المفاتيح ستنتهي هذه الحلقة. وسيحقق هذا ما تحققه وظيفة INKEY( ) تماماً، كما بينا. إلا أنه بما أن هذه العملية هي حلقة، فيمكن أن نضع داخلها أشياء أخرى.

والمثال البسيط على هذه العملية هو "صوت حركة الساعة". فقد كانت الطريقة الوحيدة باستخدام برنامج Summer'87 لعرض ساعة تصدر صوتاً خلال حالة انتظار هي ربط إيقاف مؤقت من خلال مكتبة شركة أخرى. أما في كليبر 5.x فإن كتل الشيفرة تسهل الموضوع تماماً. وتبين الكتلة البرمجية التالية سهولة تنفيذ مثل هذا الأمر:

```
bevent := { | | clock( ) }
do while ( key := inkey( ) ) == 0
   eval(bevent)
enddo
```

ويتم تقييم كتلة الشيفرة هذه بشكل مستمر طالما أنه لم يتم الضغط على أي مفتاح من لوحة المفاتيح، (أو لم تتحرك الفأرة من موضعها) وسيدهشك هذا الأداء.

لاحظ أنه من المحتمل أن يكون الأمر أفضل إذا تمت الإشارة إلى هذه الوظيفة في كتلة الشيفرة بشكل سريع نسبي بحيث نرجع وظيفة INKEY( ) ليكون متكرراً، وإلا فإن المستخدم قد يضغط على المفتاح ويضطر للانتظار عدة ثوان للحصول على جواب. ولن يكون هذا الأمر عادياً، إلا أننا إذا سمحنا بإرسال كتلة الشيفرة على أنها متغير آخر للوظيفة فستصبح عندئذ عامة.

وقت مستقطع اختياري وحادثة

والآن، وبعد أن تعرفنا على حلقة DO WHILE يمكننا إضافة مزيد من التعابير لنقصرها. ولعل أوضح تعبير هو "الوقت المستقطع". فسنؤسس متغيراً للوقت المستقطع وآخر لوقت البدء به.

```
local nstart := seconds( )
local nseconds := 30
```

ثم نضيف التعبير التالي على عبارة DO WHILE :

```
do while ( key := inkey( ) ) == 0 .and. ;
    seconds( ) - nstart < nseconds
enddo
```

ولا بد أن نضيف أخيراً اختياراً آخر بعد حلقة DO WHILE لنتأكد من كيفية خروجنا من الحلقة. ويمكننا أن نفترض أنه إذا كان KEY لا يزال صفراً، فإن هذا يعني أننا كسرنا الحلقة بسبب وجود شرط الوقت المستقطع. وتكون الخطوة التالية في مثل هذه الحالة استدعاء بعض الوظائف الأخرى كأن نستدعي وظيفة تغطية الشاشة. إلا أننا إذا أردنا أن تكون هذه الوظيفة مرنة فإنه يجب أن نسمح لكتلة الشيفرة أن تحتوي على وظيفة "وقت مستقطع". فعلى سبيل المثال يمكن أن تستدعي كتلة الشيفرة وظيفة تغلق قاعدة البيانات وتوقف البرنامج.

```
if key == 0
   if bexit != NIL
      eval(bexit)
   else
     blankscm( )
   endif
else ...
```

استخدام أمر يعرفه المستخدم لسهولة القراءة

لقد أضفنا عدة متغيرات MyInKey( ) إلا أننا قد لا نريد استخدامها جميعاً بشكل دائم. وبدلاً من محاولة تذكر ترتيب تلك المتغيرات والذي يستحسن تجنبه، فيمكن استدعاء ما قبل المعالج لتسهيل الأمر.

ونبين فيما يلي أمراً يعرّفه المستخدم لهذا الغرض:

```
#xcommand     NKEY                        ;
              [TO  <v> ]                  ;
              [TIMEOUT <t> ]              ;
              [EVENT  <e>  ]              ;
              [EXIT  <exit>  ]            ;
```

```
                              =>          ;
[ <V>  :=  }  myinkey(#<v> , <{e}>, <t> , <{exit}> )
```

ويمكن تحديد هذه الشروط بأي ترتيب تريده (أو دون ترتيب على الإطلاق). ولاحظ أن المعالج الأولي سيحول وظيفتي EVENT و EXIT إلى كتل شيفرة بشكل آلي. ويجب أيضاً ملاحظة استخدام "dumb stringity" الذي يعدّ النتيجة ("#") بحيث يحول اسم المتغير إلى مصفوفة حرفية.

والآن جميعاً معاً:

```
#xcommand INKEY                           ;
        [TO <v> ]                         ;
        [TIMEOUT <t> ]                    ;
        [EVENT <e>    ]                   ;
        [EXIT <exit>    ]                 ;
                   =>                     ;
        [<v> :=] myinkey(#<v>, <{e}>, <t>, <{exit}>)

#include "inkey.ch"

#define TEST   // to compile test stub

#ifdef TEST    // begin test stub

function test
local help1
set key K_F1 to helpme
scroll()
? "Press F1 for help... NOT!"
inkey to help1 event clock() timeout 8 exit cleanup()
return nil
static function cleanup
close data
scroll()
? "Program terminated due to inactivity..."
inkey(2)
keyboard chr(K_ESC)
```

```
return nil

static function clock
local r := row(), c := col()
@ 0, maxcol() - 7 say time()
setpos(r,c)
return nil

static function helpme
? "Sorry... no help available"
return nil
#endif    // end test stub

function myinkey(v, bevent, timeout, bexit)
local nKey := 0
local b
local start := seconds()
local mainloop := .t.
if timeout == NIL
  timeout := 600000
endif
do while mainloop
  do while ( nKey := inkey() ) == 0 .and. seconds() - start < timeout
    if bevent != NIL
      eval(bevent)
    endif
  enddo
  if nKey == 0     // we timed out of the loop
    if bexit != NIL
      eval(bexit)
    else
```

```
      blankscr3(-1)   // Grumpfish Library screen blanker
    endif
  else
    if ( b := setkey(nKey) ) != NIL
      eval(b, procname(1), procline(1), v)
      start := seconds()    // restart timeout loop counter
    else
      mainloop := .f.
    endif
  endif
enddo
return nKey
```

تغيير متغير محلي بواسطة كتلة شيفرة

والآن، بعد أن عرفنا كيف يمكن استخدام وظيفة ( )SETKEY لربط كتل الشيفرة مع ضغطات المفاتيح، لابد أن ندرك أنه لا يوجد البتة ما يمنعك من كتابة كتلة برمجة على النحو التالي:

```
#include "inkey.ch"

function main
local x := "this will be visible in the HelpMe( ) hot key function"
setkey(K_F!, { | p,1, v | helpme(p, 1, v, x) } )
// etcetera
```

وتمكن هذه الكتلة المفتاح المباشر الذي حددته بحيث يقبل المتغيرات القياسية الثلاثة لكليبر وهي: (الإجراء، ورقم السطر، واسم المتغير) إلى جانب المتغير المحلي x.

ولنفترض أن المفتاح المباشر الذي تريد استخدامه هو فقط يحتاج الاطلاع على المتغير x فقط وليس بحاجة لاستخدام المتغيرات الثلاثة القياسية الأخرى لكليبر، فيمكنك تجاوزها بكتابة كتلة برمجة باستخدام وظيفة ( )Setkey بحيث لا تقبل تلك المتغيرات.

```
setkey(K_F1, { | | helpme(x) } )
```

وإذا أردت إجراء أمر أكثر سهولة، فيمكنك إرسال متغير محلي بالإشارة فقط وبهذا تمكّنه من التغير في وضعية وظيفة المفتاح المباشر، ويبين هذا المثال التالي. نريد الحصول على متغير ما، أثناء السماح للمستخدم بفتح قائمة اختيار ما والاختيارمن مدخلات صحيحة فيها. وقد يبدو هذا الأمر بسيطاً للغاية، وهو فعلاً كذلك، إلا أن الحصول على المتغير المطلوب هو "محلي" Local، لذا فهو محدد في المجال للوظيفة التي نريد الحصول عليها فقط. وإليك حل هذا الإشكال الفني باستخدام طريقة ذكية لكتل الشيفرة.

```
#include "inkey.ch"
#include "box.ch"

function test
local mvalue := space(7), oldaltv, x, getlist := { }
if ! file("lookup.dbf")
   dbcreate("lookup", { { "LNAME", "C", 7, 0 } } )
   use lookup new
   for x := 1 to 9
       append blank
        replace lookup->lname with { "BAKER", "BOOTH", "FARLEY" , ;
             "FORCIER", "BRITTEN", "LIEF", "MEANS", "NEFF" } [x]
   next
else
  use lookup
endif
// ----- note that MVALUE is passed by reference , and that the three
// ----- standard Clipper parameters (P, L, V) are ignored
oldaltv := setkey( K_ALT_V , { | | View_Vals(@mvalue, "lname") } )
setcolor('+gr/b')
scroll( )
@ 4, 28 SAY "Enter last name: get mvalue
@ 5, 23 SAY ' (Press Alt-V for available authors) ' color '+w/b'
read
@ 7, 28 Say "You selected " + mvalue
return nil

static function view_vals( v, cfield)
local browse, column, key := 0 , marker := recno( ),
       oldscrn := savescreen(8, 35, 29, 44),
       oldcolor := setcolor("+W/RB") , oldcursor := setcursor(0),
       oldblock := setkey( K_ALT_V, NIL )       // turn off ALT-V
@ 8, 35, 19, 54 box B_SINGLE + chr(32)
browse := TBrowseDB(9, 36, 18, 43)
```

```
browse:colorSpec := '+W/RB, +W/N'
browse:addcolumn(TBColumnNew( , fieldBlock(cfield) )
go top
do while key != K_ESC .and. key != K_ENTER
    do while ! browse:stabilize( )
    enddo
    key := inkey(0)
    do case
       case key == K_UP
         browse:up( )
        case key == K_DOWN
          browse:down( )
    endcase
enddo
if key == K_ENTER
  /*
    because we passed the variable BY REFERENCE in the code block,
    any changes we make here are being made to the actual variable,
    and that is the key to this whole mess working the way it does!
*/
v := lookup->lname
endif
go marker
restscreen(8, 35, 20, 44, oldscrn)
setcolor(oldcolor)
setcursor(oldcursor)
setkey(K_ALT_V, oldblock)              // reset Alt-V for next time
return
```

وقد يعتقد بعض المبرمجين أن هذا السلوك مغاير للهدف، إذ أنه يعني أن المتغيرات المحلية يمكن رؤيتها الآن في وظائف أخرى. إلا أنه لولا هذا السلوك الشاذ نسبياً لن نستطيع الحصول باستخدام أمر GET على متغير محلي. ويجب أن تتذكر أن نظام GET في كليبر، قد كتب بالكامل باستخدام كليبر ذاته. فإذا لم تستطع، لسبب من الأسباب إرسال إشارة للمتغير في وظيفة RedModal( ) والتي توجد في ملف شيفرة المصدر GETSYS.PRG فإنك لن تستطيع تعيين القيمة للمتغير.

الوظيفة FIELDBLOCK( )

هي إحد ثـلاث وظـائف جديـدة ترجـع كتـل برمجـة "الاسـترجاع/التعيـين" أو "إحضـار مجموعة" (get-set). وتمكنك كتلة برمجة الاسـترجاع/التعيـين إمـا مـن اسـترجاع قيمـة حقل أو متغير أو تعيينه. ويعتبر هذان بمثابة القلب والروح لكليبر 5.x.

وتساعدك وظيفة FIELDBLOCK( ) على تجنب عامل الماكرو. وهي ترجع كتلة برمجة لحقل محدد. ويعتبر متغير <Field> سلسلة حرفيـة تمثـل اسـم الحقـل المشـار إليـه. ويمكنك بعدئذ إما استرجاع (الحصول) أو تعيين (تجهيز) قيمة الحقل بتقييم كتلة الشيفرة الراجعة باستخدام هذه الوظيفة. فإذا لم يكـن الحقـل موجـوداً في منطقـة العمـل النشـطة حالياً فسترجع الوظيفة "صفراً". وإن التركيب اللغوي لهذه الوظيفة كما يلي:

```
fieldblock( "fieldname" ) ==
{ | _1 | if(1_ == NIL, field->fieldname, field->fieldname := _1) }
```

ويمكنـك أت تـرى بوضـوح أنـك إذا قيمّت كتلـة الشـيفرة ولم ترسـل أي متغـير مـن المتغيرات فستؤسس على أنها "صفر" وسترجع كتلة الشيفرة القيمة الحالية لاسم الحقل. أما إذا لم ترسل متغيراً فإن اختبار IF-1 = = NIL سيفشل وستعين قيمة متغـيرك لاسـم الحقل ذاته.

ملاحظة: إذا كان الحقل <Field> الذي أرسـلته إلى هـذه الوظيفـة موجـوداً في أكثر من منطقة عمل واحدة فإن هذه الوظيفة مطابقة للحقـل الموجـود في منطقـة العمـل الحالية فقط.

وإليك مثالاً على استرجاع القيمة:

```
local bblock, mfield := "FNAME"
dbcreate("customer", { { "FNAME", "C", 10, 10 } } )
use customer new
append blank
customer->fname := "JOE"
bblock := fieldblock(mfield)
```

```
? eval(bblock)           // "JOE"
/* Note the dreaded macro alternative */
? &mfield                // slow, and simply no longer chic
```

لتعيين قيمة ما لحقل، يجب أن تقيّم كتلة الشيفرة وترسل القيمة المطلوبة على شكل متغير. مثال:

```
local bblock, mfield := "FNAME"
use customer new
bblock := fieldblock(mfield)
eval(fieldblock(mfield), "Adam")
? customer->fname
/* note the dreaded macro alternative */
replace &mfield with "Adam"     // ugh! return nil
```

الوظيفة FIELDWBLOCK(<field>, <work area>)

تشبه هذه الوظيفة سابقتها إلى حد كبير، إلا أنها تمكنك من الإشارة إلى منطقة عمل مختلفة لاسترجاع قيمة حقل أو تعيينه. وهي مفيدة بشكل خاص عند تجهيز أمر TBrowse يحتوي على حقول من أكثر من منطقة عمل واحدة.

واليك مثال على هذه الوظيفة.

```
dbcreate("customer" , { { "LNAME", "C", 10, 0 } })
dbcreate("vendor" , { { "LNAME", "C" , 10, 0 } } )
use customer new
append blank
customer->lname := "CUSTOMER1"
use vendor new
append blank
vendor->lname := "VENDOR1"
? eval(fieldwblock("LNAME", select("customer") ) ) // CUSTOMER1
? eval(fieldwblock("LNAME", select("vendor") ) ) // vendor1
? eval(fieldwblock("LNAME", select("vendor") ) ) , "Grumpfish")
? vendor->lname                                  // Grumpfish
```

ومن السهل تعيين قيمة لحقل باستخدام هذه الوظيفة كما هو الحال في سابقتها. قيّم كتلة الشيفرة الراجعة باستخدام وظيفة FIELDWBIOCK( ) وأرسل القيمة المطلوبة على أنها متغير، كما بينا في المثال السابق.

ويمكن إجراء عملية TBrowse شكل مختلف عما بيناه سابقاً باستخدام هذه الوظيفة:

```
local x, browse := TBrowseDB(3, 19, 15, 60) , column
use test new
for x := 1 to fcount( )
    column := TBColumnNew(field(x), field(x), fieldwblock(field(x), select() ) )
    browse:AddColumn( column )
next
do while ! browse:stabilize( )
enddo
```

الوظيفة MEMVARBLOCK(<memvar>)

تشبه هذه الوظيفة إلى حد كبير وظيفة FIELDWBIOCK( ) إلا أنها تعمل على متغيرات الذاكرة العامة أو الخاصة بدلاً من العمل على حقول قواعد البيانات. وترجع هذه الوظيفة كتلة الشيفرة لمتغير الذاكرة كما تم تحديده باستخدام متغير <memvar>. ثم يمكنك عندئذ إما استرجاع القيمة بتقييم كتلة الشيفرة الراجعة باستخدام أمر الوظيفة أو تعيينها. أما إذا لم يكن <memvar> موجوداً فسترجع هذه الوظيفة قيمة"الصفر" NIL.

تحذير

إذا كان متغير <memvar> ساكناً أو محلياً فإن هذه الوظيفة سترجع أيضاً القيمة "صفر" NIL لأن هذه الوظيفة لا تعمل إلا على المتغيرات "العامة" و "الخاصة" فقط.

ويبدو أنه ليس من المفيد جداً استخدام هذه الوظيفة نظراً لاستخدام إعلانات LOCALو STATIC بدلاً من PUBLIC و DRIVAR في كليبر 5.x.

توسيع أوامر كليبر باستخدام كتل الشيفرة

تستخدم كثير من وظائف كليبر كتل الشيفرة، وكلما ازدادت معرفتك بهذه اللغة استطعنا استخدام قواها الكامنة وفعالياتها في كتابة كتل برامجك المختلفة. ولعل أفضل طريقة لنتعلم الجرأة على استخدام هذه الوظائف والقوى الكامنة هي أن تبدأ باستخدام التجميع بطريقة مفتاح P/ إذ يعدّ هذا الأمر ملف إخراج مسبق المعالجة (PPO.) يحتوي على مصدر برنامجك بعد أن تم الانتهاء من ترجمة المصدر من قبل ما قبل المعالج.

ولعل أفضل مثال هو أمر استخدام الفهرسة INDEX ON وسنركز على هذا المثال من خلال الأمثلة التالية:

```
index on keyfield to indexfile
```

ويترجم هذا الأمر من قبل ما قبل المعالج على النحو التالي:

```
dbCreateIndex( "indexfile", "keyfield",  { | | keyfield} , ;
                   if(  .F.,  .T.  , NIL  ) )
```

وكما بينا لدى الحديث عن "ما قبل المعالج" فإن وظيفة ( )db CreatIndex قد أضيفت مع الإصدار 5.01 ومتغيراتها هي كما يلي:

- سلسلة حرفية تمثل اسم ملف الفهرس المراد إنشاؤه.
- سلسلة حرفية تمثل التعبير الاساسي. وسيتم إعداد هذا وجعله جزءاً من ملف لترويسة NTX.
- كتلة برمجة اختيارية تمثل التعبير الاساسي، وتقيم هذه لكل سجل في قاعدة لبيانات لإنشاء ملف فهرسة. وإذا تم تجاوز هذا المتغير فسيتم تحويل المتغير لثاني إلى كتلة برمجة يتم تقيمها.
- متغير منطقي يحدد ما إذا كان سيتم عمل فهرس متميز Unique أم لا.

وسنهتم هنا بشكل أساسي بالمتغير الثالث وهو كتلة الشيفرة، وكل ما ستفعله الآن هو إعادة التعبير الأساسي:

```
{  | | keyfield }
```

وسيكون بمنتهى البساطة بالنسبة لنا إدخال وظيفة استدعاء قبل التعبير الأساسي وتتمكن هذه الوظيفة من عرض سطر الوضعية. ولدى انتهاء عملية الفهرسة لن يكون لهذه الوظيفة أية علاقة على الإطلاق بملف الفهرسة، وسنستخدم الأمر التالي الذي يحدده المستخدم لتحسين نوعية القراءة:

```
#xcommand  INDEX  ON  <key>  TO  <(file)>  GRAPH  [<u:  UNIQUE>] ;
                            =>  dbCreateIndex(  <(file)> , <"key"> ,  ;
                            { |   | indexbar( ),  <key> } , <.u.> )
```

إن الفارق بين هذا الأمر، ومجموعة أوامر كليبر هو الكلمة الأساسية GRAPH واستدعاء الوظيفة الموجود داخل كتلة الشيفرة. لاحظ أن استدعاءات وظيفة IndexBar( ) و LASTREC( ) و RECNO( ) هي الحد الأدنى اللازم إذ أن هذا سيحسن نوعية الأداء.

```
#include "box.ch"
#define  TESTING        // to compile test stub
/*
  This assumes a database of at least 60 records... smaller
  databases should not really require visual feedback for
  the index process anyway...
*/
#ifdef TESTING   // begin test stub
#xcommand INDEX ON <key> TO <(file)> GRAPH [<u: UNIQUE>]
     => dbCreateIndex( <(file)>, <"key">, { || indexbar(), <key>}, <.u.>)

function test
local x
scroll()
```

```
? "creating test database.."
dbcreate("test.dbf", { { "NAME", "C", 2, 0 } } )
use test
for x := 1 to 10000
  append blank
  test->name := replicate(chr(x % 256), 5)
next
index on test->name to test graph
use
ferase("test.dbf")
ferase("test.ntx")
return nil

#endif  // end test stub
/*
    Function: IndexBar()
    Purpose:  Display status bar during index process
    Params:   None
    Returns:  Nada
*/
function indexbar
static nlastrec
static screen
static ngraphlen
static nspacing
local curr_rec := recno()
if nlastrec == NIL
  //--- establish NLASTREC and NGRAPHLEN variables
  nlastrec := lastrec()
  nspacing := nlastrec / 60
  ngraphlen := 0
```

```
  //----- save screen and draw initial box
  screen := savescreen(9, 8, 11, 71)
  @ 9, 8, 11, 71 box B_SINGLE + ' ' color 'w/b'
  @ 9, 33 say " Index Status "
  @ 10, 10 say replicate(chr(177), 60) color 'w/b'
  setpos(10, 10)
else
  //----- display characters only if necessary
  if ngraphlen != int ( curr_rec / nspacing )
     ngraphlen++
     dispout(chr(219), '+gr/n')
  endif
  if curr_rec == nlastrec
     //----- if we're finished, restore the screen and reset NLASTREC
     restscreen(9, 8, 11, 71, screen)
     nlastrec := NIL
     ngraphlen := NIL
  endif
endif
return nil
```

فتح قواعد بيانات وفهارس

بعد مناقشة كل من المصفوفات، وكتل الشيفرة والوظائف المتعلقة بقواعد البيانات، لنبدأ باستخدامها جميعها لإنتاج وسائل قوية لفتح كل من قواعد البيانات والفهارس. وإن هذا لن يمكننا من فتح ملفاتنا بالطريقة والوضعية التي تريدهما فقط (مشتركة، حصرية، قراءة فقط) إلا أنه يعيد إنشاء أي ملف مفقود أو ناقص، بل إنه أيضاً يملء قواعد البيانات الحديثة الإنشاء بسجلات يجب أن تكون موجودة من البداية.

ويتم حفظ كافة بيانات قاعدة المعلومات ومعلومات الفهرسة في مصفوفة واحدة تحتوي عنصراً لكل قاعدة بيانات في برنامجك.

ومع أن هذه المصفوفة تبدو معقدة بعض الشيء إلا أنها واضحة الاستعمال. وقد تستغرب وجود حلقتين للملفات الفهرسة (احدهما للتأكد من وجودها والثاني لفتحها). وقد يبدو هذا الأمر غير كافي للوهلة الأولى، إلا أنه ضروري لأن وظيفة dbCreateIndex( ) تنشط آلياً أي فهرس تنشئه بينما تغلق أيضاً أي فهارس مفتوحة.

```
#define TEST           // to compile test stub
#ifdef TEST           // begin test stub
function test
set exclusive off
OpenFiles(  ;
        { ;
          { "CUSTOMER.DBF",          ;  // database #1
                    .f.,             ;  // .T. = exclusive (default: .F.)
                     ,               ;  // .T. = readonly (default: .F.)
               "CUST",               ;  // alias to use (optional)
                     ,               ;  // RDD to use (default: DBFNTX)
     { ;
      { "FIRST",   "C",     15, 0 },    ;  // DBF structure information
```

```
    { "LAST",    "C",      20, 0 },        ; // for use by DBCREATE() in
    { "ADDRESS", "C",  50, 0 },            ; // the event that the database
    { "CITY",    "C",      25, 0 },        ; // needs to be recreated
    { "STATE",   "C",       2, 0 },        ;
    { "ZIP",     "C",       9, 0 },        ;
    { "PHONE",   "C",      10, 0 },        ;
    { "FAX",     "C",      10, 0 }         ;
   }                                       ;
    ,                                      ; // new records to be added
   {                                        ; // one subarray for each record
    { "Greg", "Lief", "2450 Lancaster Dr NE", ;
      "Salem", "OR", "97305", "5035881815",   ;
      "5035881980" }          ;
   }                      ;
    ,                     ; // array of index information
   {                      ; // <cIndexname>,<cKey>,<lUnique>
    { "NAME.NTX", "UPPER(CUST->LAST + CUST->FIRST)", .F. }, ;
    { "CITY.NTX", "UPPER(CUST->CITY)",            .F. }  ;
   }                      ;
 }                        ;
   ,                      ;
{ "VENDOR.DBF",               ;  // database #2
    ,                     ;  // .T. = exclusive  (default: .F.)
    ,                     ;  // .T. = readonly (default: .F.)
    ,                     ;  // alias to use (optional)
    ,                     ;  // RDD to use (default: DBFNTX)
   { ;
    { "COMPANY", "C", 45, 0 }, ;  // DBF structure information
    { "CONTACT", "C",  40, 0 }, ;  // for use by DBCREATE() in
    { "ADDRESS", "C",  50, 0 }, ;  // the event that the database
    { "CITY",        "C",   25, 0 }, ;  // needs to be recreated
```

```
        { "STATE",     "C",   2, 0 }, ;
        { "ZIP",       "C",   9, 0 }, ;
        { "PHONE",     "C",  12, 0 }, ;
        { "FAX",       "C",  12, 0 }  ;
       }                              ;
        ,                   ; // new records to be added
       {                    ; // one subarray for each record
        { "Grumpfish, Inc.", "Mary Gries", ;
          "2450 Lancaster Dr NE", "Salem", "OR", ;
          "97305", "5035881815", "5035881980" }  ;
       }                    ;
        ,                   ; // array of index information
       { ;                    // <cIndexname>,<cKey>,<lUnique>
        { "COMPANY.NTX", "UPPER(VENDOR->COMPANY)", .F. }
       }                    ;
     }                      ;
   }  )

inkey(0)
return nil

#endif  // end test stub
// manifest constants to delineate the structure of the array
#define DBF_NAME           1
#define EXCLUSIVE_USE      2
#define READONLY_USE       3
#define ALIAS_NAME         4
#define RDD_NAME           5
#define DBF_STRUCTURE      6
#define NEW_RECORDS        7
#define INDEX_INFO         8
```

```
#define INDEX_NAME        1
#define INDEX_KEY         2
#define UNIQUE_INDEX      3

function OpenFiles(a)
local nDbfs := len(a)
local x
local y
local n
local lMustfill
for x := 1 to nDbfs
  lMustfill := .f.
  //----- check for existence of database, recreate if necessary
  if ! file( a[x][DBF_NAME] )
    dbcreate( a[x][DBF_NAME], a[x][DBF_STRUCTURE] )
    lMustfill := .t.  // so any new records will be added below
  endif
  dbUseArea( .t.,
         a[x][RDD_NAME],
         a[x][DBF_NAME],
         a[x][ALIAS_NAME],
         if(a[x][EXCLUSIVE_USE] != NIL,
           ! a[x][EXCLUSIVE_USE], NIL),
         a[x][READONLY_USE])
  n := len( a[x][INDEX_INFO] )
  //----- first verify existence of all indexes, recreating if necessary
  for y := 1 to n
    if ! file( a[x][INDEX_INFO][y][INDEX_NAME] )
      dbCreateIndex(a[x][INDEX_INFO][y][INDEX_NAME],
           a[x][INDEX_INFO][y][INDEX_KEY],
           &("{ || " + a[x][INDEX_INFO][y][INDEX_KEY] + "}"),
```

```
            a[x][INDEX_INFO][y][UNIQUE_INDEX])
      //---- dbCreateIndex() automatically opens the newly created
      //---- index.  We must close it now because we'll open it below
      dbClearIndex()
    endif
  next
  //---- now activate the indexes
  for y := 1 to n
    dbSetIndex( a[x][INDEX_INFO][y][INDEX_NAME] )
  next
  //---- finally, add any new records that were specified
  //---- if we had to recreate the database from scratch
  if lMustfill .and. valtype( a[x][NEW_RECORDS] ) == "A"
    n := len( a[x][NEW_RECORDS] )
    //---- loop through info array and add one new record
    //---- for each element
    for y := 1 to n
      dbAppend()
      aeval( a[x][NEW_RECORDS][y], ;
            { | info, fieldnum | fieldput(fieldnum, info) } )
    next
  endif
next
return nil
```

أمر "فرّق/جمّع Scatter/Gather

لقد أضاف مجمع كليبر ثلاثة أوامر تسهل عمل المبرمج إلى حد كبير وهي:

FIELDGET( ) و FIELDPUT( ) و FIELDPOS( ) . وتسهل هذه الوظائف "التفريق" (نسخ حقول قواعد البيانات إلى متغيرات الذاكرة لتحريرها) و "التجميع" (تحديد قيم لمتغيرات الذاكرة ذاتها إلى حقول قواعد البيانات). دون الحاجة إلى استخدام الماكرو.

أمر FIELDGET(<nfield>)

يبين هذا الأمر الحقل الذي سيتم استخدامه طبقاً لإحداثياته في تركيبة ملف قاعدة البيانات. وسيرجع قيمة الحقل الذي هو موضع السؤال. مثلاً: إذا أرسلت القيمة ٢ إلى الأمر FIELDGET( )، وكان الحقل FNAME هو الحقل الثاني في تركيبة قاعدة بياناتك فإن أمر FIELDGET( ) سيرجع قيمة FNAME إلى السجل الحالي.

أمر FIELDPUT(<nfield> , <newvalue>)

يحدد أمر <nField> كسابقه الحقل الذي سيستخدم في تركيبة قاعدة البيانات. وتمثل <newvalue> القيمة التي ستعين للحقل، كما تخدم أيضاً كعمل القيمة الراجعة من أمر FIELDPUT( )، كما يبين هذا البرنامج التالي:

```
y  := fieldput(2,  "Ahmed")                // assigns "Ahmed" to 2nd field
?  y
```

وتبين القيم المدرجة أدناه روتيني فرق/جمع يستخدم الأول عدداً من الماكرو بينما يستخدم الثاني الأوامر التي نناقشها هنا.

```
function  test
```

```
local nfields, xx, ahold := { }, mfield
memvar getlist
// first create test database
dbcreate('rolodex' , {    { "FNAME",    "C",  15, 0} , ;
                          { "LNAME",    "C",  15, 0} , ;
                          { "ADDRESS",  "C",  35, 0} , ;
                          { "CITY",     "C",  30, 0} , ;
                          { "STATE",    "C",  2,  0} , ;
                          { "ZIP" ,     "C",  10, 0} , ;
use rolodex new
nfields := fcount( )
scroll( )
// first let's try it with macros

// dump all field contents to array for editing
for xx = 1 to nfields
   mfield = field(xx)
   aadd(ahold, &mfield)
   @ xx, 1 say padr(mfield, 11) get ahold[xx]
next
read
append blank

// now dump array contents to the fields of the blank record
for xx = 1 to nfields
    mfield = field(xx)
    replace &mfield with ahold[xx]
next
// now with FIELDGET( ) and FIELDPUT( )
ahold := { }  // clear out the array
// dump all field contents to array for editing
for xx = 1 to nfields
   aadd(ahold, fieldget(xx))            // lookma, no macro
   @ xx, 1 say padr(field(xx), 11) get ahold[xx]
next
read
// now dump array contents to the fields of this record
aeval(ahold, { | ele, num | FieldPut(num, ele) } )
return nil
```

ويعتبر هذا المثال بسيطاً نسبياً لأنه ينشيء اختبار قاعدة بيانات في كل مرة، إلا أنـك إذا أردت إضافة سجل إلى قاعدة بيانات تحتوي على سجلات، فبـدلاً مـن القيـام بعمليـات معقدة وطويلة لتحدد القيم الأولية لكل حقل فيمكنـك إصـدار أمـر: GO قبـل تحميـل

مصفوفة AHOLD. وإن أية محاولة للذهاب إلى أي سجل خارج حدود النطاق ستضع مؤشر السجل على LASTREC( )+1 والذي يعرف أحياناً، أو يشار إليه باسم "سجل الشبح" (Phantomreeurd).

وقد فرقنا وجمعنا في المثال السابق كافة الحقول الموجودة في قاعدة البيانات، ولكن لنفترض أنك تريد تفريق/وتجميع حقلين فقط، فهنا يأتي دور أمر FIELDROS( ) إذ أنه يرجع مكانة حقل محدد داخل ملف قاعدة البيانات مطابقة لمنطقة العمل.

أمر FIELDPOS(<cField>)

إن اسم <FIELD> هو اسم الحقل المطلوب في منطقة العمل الحالية. وإن أمر FIELDPOS( ) يرجع مكان إحداثيات الحقل في تركيبة ملف قاعدة البيانات المتعلقة بمنطقة العمل الحالية. وإذا لم يتم العثور على ذلك الحقل في تلك المنطقة فإن الأمر FIELDPOS( ) سيرجع إلى الرقم صفر. وتعتمد القوائم المبينة أدناه على أمر FIELDPOS( ) للتفريق والتجميع لحقلين من قاعدة البيانات ROLODEX:

```
function main
local fields_  := { "LNAME" , "ADDRESS" }
local ahold_  :=  { }
local nfields  := len(fields_)
local x
use rolodex new
for x  = 1 to nfields
   aadd(ahold_,  fieldget(fieldpos(fields_[x] ) )  )
   @ x,  1 say  padr(fields_[x]  +  ":" , 12)  get  ahold_[x]
next
read
// now dump  array  contents to the  fields of  this  record
aeval(ahold_,  {  |  ele,  num  |  fieldput(fieldpos(fields_[num]),  ele)  } )
return nil
```

وكما أوضحنا أثناء مناقشة كتل الشيفرة، فإذا كان لديك سلسلة حرفية تحتوي على اسم حقل، فلن تعود بحاجة إلى تطبيق عامل الماكرو لتنشيط قيمة الحقل، بل يمكنك

إرسال تلك السلسلة الحرفية إلى وظيفة FIELDBLOCK( ) والتي سترجع بدورها كتلة برمجة تحتوي على إشارة إلى الحقل. ثم يمكنك بعدئذ تقييم كتلة الشيفرة الناتجة لاسترجاع (أو تعيين) قيمة الحقل.

وقد ترغب أيضاً باستخدام الأوامر الثلاثة السابقة بدلاً من استخدام FIELDBLOCK( ). لماذا؟ لأنها أسرع، كما يبين المثال التالي:

```
function test
local fieldname := "NAME" , x, y, f, start
local b := fieldblock(fieldname)
dbcreate('customer', { { "NAME', "C", 20, 0 } } )
use customer
start := seconds( )
for x := 1 to 1000
     y := eval(b)
next
? "FIELDBLOCK:" , seconds( ) - start
inkey(0)
f := fieldpos(fieldname)
start := seconds( )
for x := 1 to 1000
     y := fieldget(f)
next
? "FIELDPOS( ): " , seconds( ) - start
inkey(0)
use
ferase('customer.dbf')
return nil
```

فقد يستغرق تنفيذ هذه الحلقة وقتاً أقل باستخدام أمر FIELDGET( ) بدلاً من استخدام FIELDBLOCK( ) الذي يتطلب ضعف الزمن تقريباً.

التحويل إلى نظام التشغيل DOS باستخدام الرابط BLINKER 2.0

إن الرابط Blinker هو لغة ربط مستقلة ديناميكية ، صممت خصيصاً لتستخدم مع كليبر Summer'87 وتطبيقات كليبر 5.x. فهي سريعة جداً بشكل لا يصدق، كما تقدم ربطاً متزايداً بشكل مذهل. وتعتبر قواعدها اللغوية مماثلة تماماً للرابط Rtlink. كما يقدم هذا الرابط العديد من وظائف البرامج المساعدة والمفيدة في تطبيقات كليبر ، بما في ذلك "تخزيم الذاكرة" memory packing. كما يتيح لك هذا البرنامج أيضاً وضع الأرقام المتسلسلة ومعلومات بيئة كليبر في الملف التنفيذي الخاص بك مباشرةً.

كما أضافت النسخة الحديثة من هذ البرنامج عدداً من الوظائف التي تمكنك من نقل برامجك إلى نظام التشغيل بحيث تتمكن من تحرير الذاكرة بأكملها تقريباً ، والتي كانت متوفرة قبل تحميل برنامجك!. ويتم هذا بالتقاط صورة عن الحالة الراهنة لما عليه ذاكرة جهازك وكتابة هذا كله ، أو بعضه في ملف على القرص الصلب.

ولعل السبب الأساسي في مناقشة هذا الموضوع أن كثيراً من المبرمجين الذين يطورون برامجهم باستخدام الرابط Blinker لايعرفون عن نظام التحويل والاستبدال هذا ووظائفه المتعددة. وتعتبر هذه الوظائف رائعة إذ أنها تعطي مستخدمي برامجك DOS shell وهي طبقة خارجية يمكن التعامل معها إما من خلال استخدام مفتاح مباشر أو من خلال قائمة اختيارات. ومن ناحية أخرى ، فإذا كنت تعتقد أن مستخدمي برامجك لايمكنهم التعامل مباشرةً مع نظام التشغيل ، فيمكنك إعداد برنامج صغير باستخدام كليبر ، يستخدم على أنه قائمة اختيارات أمامية. ثم تستخدم وظائف الانتقال والاستبدال لاستدعاء أي برنامج آخر من قائمة الاختيارات مباشرة ، بحيث تحجب المستخدمين تماماً عن رؤية مؤشر نظام التشغيل.

ويبين المثال التالي كيفية إعطاء مستخدمي برامجك قشرة خارجية للتعامل مع نظام التشغيل إما باستخدام برنامج Blinker 2.0 أو إصدار Overlay 3.5. وقد كتب هذا البرنامج لاستخدام Blinker ويمكن إعادة كتابته لاستخدام Overlay:

```
Clipper dosshell /dOVERLAY
```

وتأكد من ربطه بـ: OVERCL.LIB (متوفر مع Overlay).

```
#include "inkey.ch"
function dosshell
local programname := "DOSSHELL"
local lAlreadyin := .f.
#ifndef OVERLAY
  lAlreadyin := SwpGetPID(programname)
#else
  x := 0
  do while ! empty(cfile := O_GetID(++x)) .and. ;
        ! ( lAlreadyin := (cfile == programname) )
  enddo
#endif
if lAlreadyin
  alert( programname + " is already active!", ;
       "If you wish to resume, type EXIT at the DOS prompt" )
  return .f.
endif

//------ establish ALT_F10 as DOS shell key for entire program

set key K_ALT_F10 to shellout

do while lastkey() != K_ESC
  ? "Press ALT-F10 for DOS or ESC to quit"
  wait
enddo
return nil


function shellout(programname)
```

```
local oldblink := setblink()
gfsaveenv(.t., 1, 'w/n')
setpos(0,0)
scroll()

#ifndef OVERLAY

SwpSetPID("DOSSHELL")
SwpSetEnv("PROMPT=Type DOSSHELL to attempt to re-enter$_Type EXIT to
return$_$P$G")
if ! SwpRunCmd("", 0, "")
   alert("Cannot shell to DOS... Blinker error code " + ;
         ltrim(str(SwpErrMaj())) )
endif

#else

O_SetID(programname)
if ! Overlay('', 0, '', 'PROMPT=On temporary hiatus from Shell test' + ;
                  '$_Type DOSSHELL to attempt to re-enter' +     ;
                  '$_Type EXIT to return$_$P$G')
   alert("Cannot shell to DOS... Overlay() error code " + ;
         ltrim(str(O_ErrMajor())) )
endif
#endif
setpos( maxrow(), maxcol() )
setblink(oldblink)
gfrestenv()
return nil
```

وظائف تستحق الإشارة إليها:

- الوظيفة:

SWPRUNCMD(<cProgram>,<nMemory>,<cRunPath>,<cTempPath>)

هذه وظيفة تحويل أساسية، وتعتبر كافة المتغيرات فيها اختيارية. وكما أشرنا سابقاً، يمكن سلسلة كافة البرامج معاً بتحديد المتغير الأول <cProgram> ويكون هذا هو البرنامج المراد تشغيله، وإذا لم تحدد هذا الاسم ، فإن هذه الوظيفة ستحمل نسخة جديدة من ملف COMMAND.COM (وهذا ماحدث في مثالنا).

المتغير <nMemory> : هي مقدار الذاكرة الواجب تحريرها، فإذا حددت صفراً (٠) ، فستحاول هذه الوظيفة تحرير أكبر قدر ممكن من الذاكرة (غالباً يترك برنامجك أثر لايتعدى أكثر من ١٢ كيلوبايت فقط من الذاكرة).

المتغير <cRunPath> : هو اسم المسار الذي يجب التحويل إليه قبل تشغيل "البرنامج". وإذا لم تحدد هذا المسار فستتم كل الأعمال داخل الدليل الحالي لنظام التشغيل.

المتغير <cTempPath> : هو اسم المسار الذي سيستخدم للاحتفاظ بصورة الذاكرة RAM ، فإذا كانت لديك أسطوانة تحتوي على هذا الملف فيستحسن تحديدها لإسراع عملية التحويل.

- الوظيفة SWPRUNCMD( ) ، وهي بهذا الشكل دون ذكر المتغيرات ترجع قيمة منطقية تشير ما إذا كانت العملية قد تمت بنجاح أم لا. فإذا أرجعت هذه الوظيفة قيمة "غير حقيقي" فيمكن عندئذ استخدام كل من وظيفتي BLISWPMAJ( ) و BLISWPMIN( ) لتحديد رقم رسائل الأخطاء.

- الوظيفة:

SWPSETPID(<cProgram>)

تمكنك هذه الوظيفة من تهيئة اسم العملية الأساسية التي بدأت بها. وإذا استخدمت هذه الوظيفة مع وظيفة SWPGETPID() فإنها تحول دون السماح للمستخدم

بتشغيل برنامج تفرع من غلافه الخارجي (وهذا ماتريد فعلاً تجنبه). وتجـدر الإشـارة إلى ضرورة الانتباه إلى أن الطول الأقصى لهذا المتغير <cProgram> هو ٣١ حرفاً.

• الوظيفة:

SWPGETPID(<cProgram>)

تستخدم هذه الوظيفة لتحديد ما إذا كان يتم تشـغيل البرنـامج حاليـاً أم لا. ولابـد مـن تهيئة اسم البرنامج مسبقاً باستخدام وظيفة ( )SWPSETPID. وترجع هـذه الوظيفـة قيمة منطقية حقيقية إذا كان اسم البرنامج مطابقاً تماماً للبرنامج المطلوب. ويجب الانتبـاه أيضاً إلى أن الطول الأفقي لاسم <cProgram> هو ٣١ حرفاً.

• الوظيفة:

SWPSETENV(<cString>)

هذه الوظيفة جديدة في الرابط BLINKER ، وتمكن هذه الوظيفة العمليـة الأساسية من القيام بعمليات إضافية ، أو تعديل ، أو حذف لمتغيرات البيئـة. وإن الطـول الأقصـى لهذه السلسلة هو ٥١٢ بايت ، ويمكن أن تحتوي السلسلة على متغـيرات يجب ضبطهـا أو تغييرها. ويجب فصل هذه المتغيرات برمز آسكي ٢٥٥.ويجب الانتباه إلى أنه لاحاجـة لوضع CHR(255) في نهايـة السلسـلة، وإذا احتجـت إلى كتابـة القيمـة الحرفيـة هـذه داخل السـلسلة ، فيجب تحديد اثنتينفي السطر.

وقـد غيرنـا في مثالنـا أعـلاه مؤشـر DOS بحيـث يظهـر علـى النحـو التـالي: "<Type exit to return>" ، متبوعاً باسم المسار الحالي pg ، كمـا حذفنـا متغير بيئة كليبر.

ملاحظة
يتم ضبط متغييرات البيئة هذه فقط عند تنفيذ عملية التحويل ولاتنطبق إلا علـى نسـخة البيئة الفرعية فقط. أي: أن يكتب المستخدم "EXIT" للعودة إلى البرنامج الذي خـرج من غلافه ، وستعود البيئة إلى ماكانت عليه سابقاً.

لم نهدف من هذا العرض للرابط BLINKER 2.0 أن نقدم عرضاً شاملاً وكاملاً عن إمكانيات هذا البرنامج ووظائف التحويل فيه. ولابد من الاستئناس بالأدلة المرافقة للبرنامج ذاته للتعرف على كافة الوظائف التي يمكن القيام بها.

الخلاصة

تأمل دار الميمان للنشر والتوزيع أن يحتفظ الإخوة المبرمجون العاملون على إعداد برامجهم وتطويرها باستخدام كليبر بهذا الكتاب ، ويحاولوا الاستفادة منه إلى الحد الأقصى.

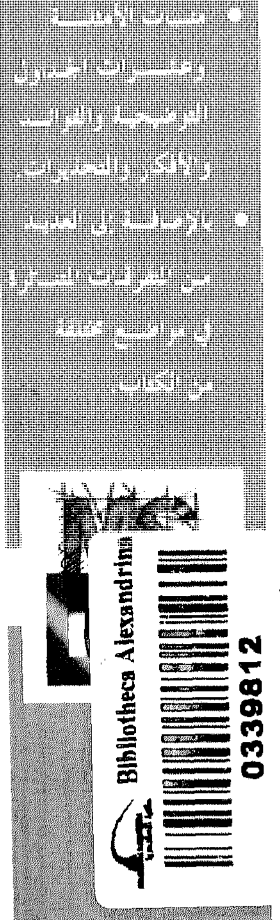
ونأمل كذلك أن توافونا بتعليقاتكم ، وآرائكم ، ومقترحاتكم واستفساراتكم التي سنأخذها قطعاً بعين الإعتبار والتقدير ، وسنكون شاكرين ومقدرين لكم كريم تعاونكم وتجاوبكم مع كتبنا.

إن كليبر هو لغة برمجة لانهائية الحدود والإمكانيات (حتى مع وجود بيئة النوافذ) التي يمكنك الاستفادة منها إلى حد كبير بحيث يمكنك تطوير برامج ذات نوعية عالية الجودة وقدرات متميزة ، ووظائف رائعة قوية ، وشاشات أنيقة وجذابة.

محاضرات كليبر

- الكتاب الوحيد في العالم العربي الذي يشرح كليبر 5.2 ، بالإضافة إلى الكم الهائل من المعلومات التي يشرحها الكتاب من خلال أجزائه الثلاث.
- تم تقسيم الكتاب إلى ثلاث أجزاء هي: مقدمة البرمجة ، أساسيات البرمجة ، البرمجة المتقدمة.
- شرح لمعظم أوامر وتعليمات ووظائف كليبر الأساسية للإصدار 5.2.
- جزء خاص عن مقدمة البرمجة بلغة كليبر ماهي؟ وكيف يمكنك الاستفادة منها؟
- تصميم وإنشاء وكتابة أقوى التطبيقات الاحترافية باستخدام لغة كليبر 5.2.
- فصل موسع لطريقة استخدام برنامج كشف الأخطاء **Debugger** بأسلوب سهل ومبسط.
- كما خصص مقدار كبير من الجزء الثالث للحديث عن البرمجة باستخدام **Tbrowse** و **TBColumn** ومزاياها المفيدة في استعراض قواعد البيانات.
- كما تم شرح كتلة الشيفرة بأسلوب سهل ، يجعل من هذه التقنية الجديدة في كليبر مريحة وسهلة الاستخدام.
- كما خصص فصل للحديث عن استراتيجيات عمل الشبكة نوفيل مع كليبر 5.2
- فصل كامل للحديث عن مفاتيح المجمع والرابط.
- شرح للمعالج الأولي **Preprosessor** وملفات الترويسة والموجهات وغير ذلك.
- فصل كامل لشرح طريقة إعلان المتغيرات بجميع أنواعها بأسلوب سهل وميسر.
- فصل كامل لشرح طريقة استخدام المصفوفات وكذلك الوظائف المتعلقة بها.
- شرح طريقة التحويل إلى نظام التشغيل **MS-DOS** باستخدام الرابط **BLINKER 2.0**
- شرح لطريقة تحويل برامجك من شيفرة المصدر **source code** إلى برامج قابلة للتنفيذ **EXE.** تعمل بشكل مستقل.
- فصل كامل يوضح طريقة تصميم واجهة المستخدم والأدوات التي يوفرها كليبر للقيام بهذه المهمة في توفير شكل جمالي ومريح للمستخدم.

- الربط باستخدام **RTLINK** وكذلك **BLINKER**
- جزء كامل يوضح أساسيات البرمجة باستخدام كليبر 5.2
- جزء كامل للحديث عن البرمجة المتقدمة باستخدام **Tbrowse** وكذلك **TBColumn**
- فصل كامل عن كتل الشيفرة **Code Blocks**
- مئات الأمثلة وعشرات الجداول التوضيحية والفوائد والأفكار والتحذيرات.
- بالإضافة إلى العديد من الملاحظات المتناثرة في مواضع مختلفة من الكتاب.





مطبعة سفير تلفون ٤٩٨٠٧٨٠ - ٤٩٨٠٧٧٦ ★ الرياض